Towards a Formal Model of the X86 ISA

Warren A. Hunt, Jr. Matt Kaufmann

Department of Computer Science
University of Texas at Austin
Austin, TX 78712, USA
{hunt,kaufmann}@cs.utexas.edu

Abstract

We present a preliminary formalization of a subset of the x86 instruction set. Our model is written in the logic of the ACL2 theorem prover. It can be executed as a Lisp program on concrete data, which provides the capability to validate the model against results delivered by actual x86 processors. We demonstrate how bugs in our model can also be eliminated by using the ACL2 prover to verify *guards* (semantic preconditions) for our functions.

Contents

1 Introduction **2**

2 Representation of State **2**

3 Instruction Interpreter **3**
3.1 Fetch and decode 4
3.2 Instruction step functions 6

4 Guard Verification **7**

5 An ACL2-driven Regression Test of Efficient Execution **8**

6 Conclusion and Future Work **9**

7 APPENDIX: Input Files **10**

8 ========== File x86-64/constants.lisp ========== **11**

9 ========== File x86-64/misc-events.lisp ========== **28**

10 ========== File x86-64/operations.lisp ========== **38**

11 ========== File x86-64/x86-state.lisp ========== **43**

12 ========== File x86-64/read-over-write-proofs.lisp ========== **106**

13 ========== File x86-64/read-over-write.lisp ========== **121**

14 ========== File x86-64/x86-utils.lisp ========== **123**

15 ========== File x86-64/x86-general.lisp ========== 140

16 ========== File x86-64/x86.lisp ========== 171

17 ========== File tools/script.lisp ========== 190

18 ==== File tools/populate-stobj-with-prog-bytes.lisp ==== 192

19 ========== File tools/fib-addr-byte.lisp ========== 193

1 Introduction

We have developed a formal model of a subset of the user-mode x86 instruction set architecture (ISA). In this paper we outline briefly both our model and an example that illustrates its execution. We also include an appendix containing the entire model and a testbench for the example. This note serves as a companion to a separate paper [2] that focuses on the underlying memory model.

The ACL2 system [3] is an interactive theorem prover that has been used to model systems and to prove theorems about those models. "ACL2", which can be read as "ACL2^2", stands for "A Computational Logic for Applicative Common Lisp". As the name suggests, the logic of ACL2 is based on an applicative (purely functional) subset of the Common Lisp programming language [5]. The ACL2 home page [4] contains links to many papers describing applications of ACL2. The present paper assumes familiarity with the prefix notation of Lisp, and we hope that ACL2-specific constructs used here can be understood from context.

Our preliminary x86-64 ISA model is written in ACL2 as an executable state machine interpreter. (We have used a development copy of ACL2 circa May, 2012.) All functions implementing this model have been *guard-verified*: that is, it is proved that every function call is made on arguments that satisfy the input precondition for that function. We discuss the role of guard verification in Section 4 below.

Our model has several limitations, especially including the following.

- Only the following instructions are supported: ADC, ADD, AND, CALL, CMP, HLT, INC, JCC, JMP, LEA, LEAVE, MOV, NOP, OR, POP, PUSH, RET, SBB, SUB, TEST, and XOR.
- Our model has no support for supervisor mode, exceptions, or interrupts. If it encounters an unimplemented or unknown opcode, the interpreter immediately returns an "error state".
- Only 64-bit mode is supported (though our model is structured to enable support for other modes).
- It is assumed that effective addresses are virtual addresses (though work is in progress to add x86 idioms that support virtual memory).

However, our model is sufficient to run an x86 "Fibonacci" binary program and get the expected result; see Section 5.

In the next section we discuss our x86-64 state representation, which includes a memory model. Then in Section 3 we present our instruction interpreter for the x86-64 ISA, followed by Section 4, which presents comments on the utility of guard verification. We say a few words about an ACL2-driven regression test in Section 5. We conclude with remarks about future directions.

The source files for this work are included in an appendix. Detailed documentation and explanation may be found as comments in these files. This paper can be viewed as a very brief introduction to our model and proofs; the complete details are given in the appendix.

2 Representation of State

We represent the x86 machine state using an ACL2 single-threaded object, or stobj [1]. Specifically, the ACL2 `defstobj` event displayed below introduces the stobj, `x86-64`. The first field of `x86-64` — `rgf`, below

— is the register file, which includes registers RAX, RBX, and so on, with 16 64-bit registers in all. The most complex part of our x86 machine state representation is the two-level memory model, designed for efficient reasoning as well as both time and space efficiency; it is described elsewhere [2].

```
(defstobj x86-64
   ;; register file: an array of 16 quadwords
  (rgf :type (array (unsigned-byte 64)
                    (*m86-64-reg-names-len*))  ; 16
       :initially 0
       :resizable nil)
   ;; the program counter
  (rip :type (unsigned-byte 64)
       :initially 0)
   ;; the eflags register
  (flg :type (unsigned-byte 64)
       :initially 0)
   ;; fields elided here
   ;; the memory model (see  [2])
  (mem-table ...)
  (mem-array ...)
  (mem-array-next-addr ...)
   ;; The state of the ACL2 model.  This flag is not part of the X86
   ;; processor; it is used to signal problems with model state, such
   ;; as the processor is halted.  While this flag is NIL, the processor
   ;; model is OK; otherwise, the flag indicates (part of) the problem.
  (ms :type t  ; Any object can be placed in the  ms field
      :initially nil)
  ...
)
```

3 Instruction Interpreter

In this section we omit declarations, including guard specifications. We also omit proofs. All such details may be found in the appendices.

Our instruction interpreter takes and returns an `x86-64` state object, in the usual style for ACL2-based interpreters: take a step and recur, until encountering an error or halt state or running out of "time". We start with the definition of the interpreter, then "drill down" the call tree to give some details of how we execute x86 instructions.

```
(defun x86-64-run (n x86-64)
  (cond ((ms x86-64)  ; model state indicates halt or error
         x86-64)
        ((zp n)  ; out of ''time''
         (let ((ctx 'x86-64-run))
           (!!ms-fresh :timeout t)))  ; update  ms to show a timeout
        (t  ; else take a step and recur
         (let ((x86-64 (x86-64-step x86-64)))
           (x86-64-run (1- n) x86-64)))))
```

The step function executes a single instruction. It first attempts to decode the instruction into suitable fields. An error indicator, `erp`, indicates when not `nil` that we should stop the interpreter by putting it into the model state (`ms`). If `erp` is `nil` then we execute a single instruction according to the macro `x86-64-step-cases`, discussed in Subsection 3.2 below.

```
(defun x86-64-step (x86-64)
  (mv-let
   (erp p1 p2 p3 p4 rex opcode ModR/M sib displacement immediate operand-nbytes
        ModR/M-p ibytes)
   (x86-64-step-fetch-decode x86-64)
   (cond (erp (!ms erp x86-64))
         (t (x86-64-step-cases)))))
```

Key properties of the step and run functions are that they preserve our invariant, which states that the two-level memory structure is well-formed [2].

```
(defthm x86-64p-x86-64-step
  (implies (and (x86-64p x86-64)
                (not (nth *ms* x86-64)))
           (x86-64p (x86-64-step x86-64))))


(defthm x86-64p-x86-64-run
  (implies (x86-64p x86-64)
           (x86-64p (x86-64-run n x86-64))))
```

We now consider the fetch/decode process and then turn to the individual instruction step functions.

3.1 Fetch and decode

The fetch/decode process is defined as follows. The `b*` form below sequentially binds variables before ultimately returning the final `mv` expression. In particular:

- `(x86-64-fetch x86-64)` binds two values: an error indicator, `erp`, and an "instruction tail", `itail`, which is a number representing 15 bytes from the instruction stream. Note that although instructions are of variable length, they must respect the architectural limit of 15 bytes (indeed, `erp` is non-`nil` otherwise); hence, this `itail` contains enough bytes to decode the current instruction.
- The call of `x86-64-decode` below splits the current instruction (contained in `itail`) into fields that are suitable for execution of individual instructions.

```
(defun x86-64-step-fetch-decode (x86-64)
  (b* ((ctx 'x86-64-step)
       ((mv erp itail)
        (x86-64-fetch x86-64))
       ((when erp)  ; early exit: return an error
        (mv (!ms-erp :rip (rip x86-64))
            nil nil nil nil nil nil nil nil nil nil nil nil nil))
       (64-bit-modep (64-bit-modep x86-64))
       (cs-dp (cs-dp x86-64))  ; actually is always true at this stage of development
       ((mv erp p1 p2 p3 p4 rex opcode ModR/M sib displacement immediate
            operand-nbytes ModR/M-p ibytes)
        (x86-64-decode itail 64-bit-modep cs-dp *decode-immediate-ar*
                       *decode-operand-nbytes-ar*))
       ((when erp)
        (mv (!ms-erp :decode-error t :rip (rip x86-64))
           nil nil nil nil nil nil nil nil nil nil nil nil nil))
       ((when p2)  not yet implemented
        (mv (!ms-erp :prefix-p2 p2 :rip (rip x86-64))
            nil nil nil nil nil nil nil nil nil nil nil nil nil)))
      (mv nil p1 p2 p3 p4 rex opcode ModR/M sib displacement immediate
          operand-nbytes ModR/M-p ibytes)))
```

The fetch function calls function `rm128` to read 16 bytes of memory from the current instruction pointer. If those bytes would extend past the top address of memory, then we return an error state. Otherwise we mask out the top byte. Of course, we could have read 15 bytes instead of 16 bytes and avoided the mask (and had a slightly weaker check for going past the top of memory), but at this stage of development we prefer the clarity of using a single read.

```
(defun x86-64-fetch (x86-64)
  (b* ((ctx 'x86-64-fetch)
       (rip (rip x86-64))
       ((when (>= (+ rip 16) *2^48*))
        (mv (!ms-erp-fresh :rip rip)
            0)))
      (mv nil
          (logand (rm128 rip x86-64)
                  #ux00ffffff_ffffffff_ffffffff_ffffffff))))
```

The decoder is the composition of decoders for the various instruction fields. It returns an error state if the decoding process requires more than the architectural limit of 15 bytes. The fields returned are as follows. All bit vectors are treated as unsigned integers except for the displacement.

- `prefix`: up to 4 bytes
- `rex`: 0 if not present, else 1 byte
- `opcode`: up to 3 bytes
- `ModR/M`: 0 if not present, else 1 byte
- `SIB`: nil if not present, else 1 byte
- `displacement`: up to 4 bytes, representing a signed integer
- `immediate`: up to 8 bytes

Some auxiliary information, useful for defining instruction execution, is also returned.

- `ibytes`: number of bytes in the instruction
- `operand-nbytes`: number of bytes in an operand

```
(defun x86-64-decode (instr 64-bit-modep cs-dp decode-immediate-ar
                            decode-operand-nbytes-ar)
  (b* ((ctx 'x86-64-decode)
       ((mv erp p1 p2 p3 p4 itail ibytes)
        (x86-64-decode-prefix instr 0))
       ((when erp)
        (mv (!ms-erp :instr instr)
            0 0 0 0 0 0 0 0 0 0 0 0 0 0))
       ((mv rex itail ibytes)
        (x86-64-decode-rex itail ibytes 64-bit-modep))
       ((mv opcode itail ibytes)
        (x86-64-decode-opcode itail ibytes))
       (ModR/M-p
        (x86-64-decode-ModR/M-p opcode))
       ((mv ModR/M itail ibytes)
        (cond (ModR/M-p
               (get-instruction-byte itail ibytes))
              (t (mv 0 itail ibytes))))
       ((mv sib itail ibytes)
        (x86-64-decode-sib itail ibytes ModR/M))
       ((mv displacement itail ibytes)
        (x86-64-decode-displacement itail ibytes ModR/M))
       (opcode-ext (mrm-reg ModR/M))
       (operand-nbytes
        (x86-64-decode-operand-nbytes p3 rex opcode opcode-ext cs-dp
                                      decode-operand-nbytes-ar))
       ((mv immediate ibytes)
        (x86-64-decode-immediate itail ibytes operand-nbytes opcode
                                 decode-immediate-ar))
       ((when (> ibytes 15))
        (mv (!ms-erp-fresh :instr instr)
            0 0 0 0 0 0 0 0 0 0 0 0 0 0)))
      (mv nil p1 p2 p3 p4 rex opcode ModR/M sib displacement immediate
          operand-nbytes ModR/M-p ibytes)))
```

For individual decoding functions called above, such as `x86-64-decode-prefix` see the appendix for file `x86-general.lisp`.

3.2 Instruction step functions

We saw above that a macro `x86-64-step-cases`, is invoked to generate (code for) the individual instruction step functions. That macro expands as follows.

```
(CASE OPCODE
      ((195)
       (X86-64-STEP-RET P1 P2 P3 P4 REX
                        OPCODE MODR/M SIB DISPLACEMENT IMMEDIATE
                        OPERAND-NBYTES MODR/M-P IBYTES X86-64))
      ((80 81 82 83 84 85 86 87 104 106)
       (X86-64-STEP-PUSH P1 P2 P3 P4 REX
                         OPCODE MODR/M SIB DISPLACEMENT IMMEDIATE
                         OPERAND-NBYTES MODR/M-P IBYTES X86-64))
      ...  ; other cases omitted here
      (OTHERWISE (X86-64-STEP-UNIMPLEMENTED OPCODE X86-64)))
```

We look in depth at our implementation of the return (RET) instruction, `x86-64-step-ret`. It is defined by the following macro.

```
(defstep ret
  (b* ((rsp (rgfi *mr-rsp* x86-64))
       (new-rsp (+ rsp 8))
```

```
      ((when (> new-rsp *2^48*))
       (!!ms-fresh :rsp rsp))
      (tos8 (rm64 rsp x86-64))
      ((when (> tos8 *2^48-16*))  ; fail now instead of at next fetch
       (!!ms-fresh :rsp rsp :tos8 tos8))
      (x86-64 (!rgfi *mr-rsp* new-rsp x86-64)))
     (!rip tos8 x86-64)))
```

This macro expands to a definition (as before, declarations omitted here) and a theorem. The theorem is key for proving that our memory model invariant is preserved, theorem `x86-64p-x86-64-step` above. The definition is straightforward, increasing the RSP by 8 and setting the instruction pointer (RIP) to the top of the stack, i.e., to the value TOS8 stored at the old RSP.

```
(PROGN
 (DEFUN X86-64-STEP-RET (P1 P2 P3 P4 REX
                            OPCODE MODR/M SIB DISPLACEMENT IMMEDIATE
                            OPERAND-NBYTES MODR/M-P IBYTES X86-64)
   (LET ((CTX 'X86-64-STEP-RET))
        (DECLARE (IGNORABLE CTX))
        (B* ((RSP (RGFI *MR-RSP* X86-64))
             (NEW-RSP (+ RSP 8))
             ((WHEN (> NEW-RSP *2^48*))
              (!!MS-FRESH :RSP RSP))
             (TOS8 (RM64 RSP X86-64))
             ((WHEN (> TOS8 *2^48-16*))
              (!!MS-FRESH :RSP RSP :TOS8 TOS8))
             (X86-64 (!RGFI *MR-RSP* NEW-RSP X86-64)))
            (!RIP TOS8 X86-64))))
 (DEFTHM
   X86-64P-X86-64-STEP-RET
   (IMPLIES
    (FORCED-AND
     (DECODED-INSTRUCTION-P P1 P2
                            P3 P4 REX OPCODE MODR/M SIB DISPLACEMENT
                            IMMEDIATE OPERAND-NBYTES MODR/M-P)
     (RET$OPCODEP OPCODE)
     (NATP IBYTES)
     (X86-64P X86-64))
    (X86-64P (X86-64-STEP-RET P1 P2 P3 P4 REX OPCODE MODR/M SIB
                              DISPLACEMENT IMMEDIATE OPERAND-NBYTES
                              MODR/M-P IBYTES X86-64))))
 (IN-THEORY (DISABLE X86-64-STEP-RET)))
```

4 Guard Verification

In this section we bring guards into the discussion, and show how they can be used to discover specification bugs. We use as an example the form `(defstep ret ...)` above — specifically, the following code from that definition, which causes early exit from its `b*` form.

```
      ((when (> new-rsp *2^48*))
       (!!ms-fresh :rsp rsp))
```

Suppose we hadn't thought to include this code in the above `defstep` form. Without that code, ACL2 reports the following error in processing the corresponding definition: "The proof of the guard conjecture for X86-64-STEP-RET has failed." The summary printed by ACL2 shows some failed proof goals, the first of which is as follows. The large number on the last line is 2^{48}. The `EXTRA-INFO` hypothesis would normally be omitted, but in this case we inserted a debug declaration, `(defstep ret (declare (xargs :guard-debug t)) ...)`.

```
Subgoal 7
(IMPLIES (AND (X86-64P X86-64)
              (RET$OPCODEP OPCODE)
              (DECODED-INSTRUCTION-P P1 P2
                                     P3 P4 REX OPCODE MODR/M SIB DISPLACEMENT
                                     IMMEDIATE OPERAND-NBYTES MODR/M-P)
              (<= 0 IBYTES)
              (INTEGERP IBYTES)
              (EXTRA-INFO '(:GUARD (:BODY X86-64-STEP-RET))
                          '(RM64 RSP X86-64)))
         (< (RGFI 4 X86-64) 281474976710656))
```

The `EXTRA-INFO` term is a hint that the problem is with a call of `rm64` at address `rsp`. Recall the next `b*` binding after the deleted `when` clause above.

```
       (tos8 (rm64 rsp x86-64))
```

The guard for function `rm64` requires that we can read 8 bytes, which is the condition `rsp` $+7 < 2^{48}$ in this case. The proof output above suggests the missing `when` clause so that we return an error state when this guard condition fails, that is: `(> new-rsp *2^48*)`.

5 An ACL2-driven Regression Test of Efficient Execution

A test driver for our preliminary x86 ISA interpreter, file `tools/script.lisp` is included in the appendix, where it is followed by two supporting files. The driver runs our interpreter on x86 binary obtained from a C source program that computes numbers in the Fibonacci sequence. To get a sense of the efficiency of our interpreter, we first load the driver to do our default test.

```
(ld "../tools/script.lisp")
```

Then we try a larger input than the one used for our default test: 20 instead of 8.

```
ACL2 P>(run-fib 20 x86-64)
; (X86-64-RUN-STEPS (@ XRUN-LIMIT) ...) took
; 0.52 seconds realtime, 0.51 seconds runtime
; (100,446,464 bytes allocated).
(fib 20) was correctly computed as 6765 (270452 steps)
NIL
ACL2 P>
```

When we divide the number of instructions, 270452, by 0.52 seconds realtime, we see that we have executed at a rate of 520,100 instructions per second. This run was done using a modern 3.5 GHz Intel processor.

6 Conclusion and Future Work

We believe that our initial experiments demonstrate the feasibility of defining an accurate formal model for a substantial user-level subset of the x86 ISA. With execution speed observed exceeding 500,000 instructions per second, we have the capability to validate our model by comparing its results against results delivered by actual x86 processors.

We have identified performance bottlenecks due to the construction of large integers by our interpreter. Future work may organize the memory using an array of bytes instead of an array of quadwords. We also plan to use our model to prove properties of x86 binary programs.

Acknowledgements

This material is based upon work supported by DARPA Contract N66001-10-2-4087, with partial support from the National Science Foundation under Grant No. EIA-0303609. We thank Shilpi Goel for assisting in development of the testbench, in particular by providing tool support for parsing x86 binaries together with the resulting binary (file `tools/fib-addr-byte.lisp`) and helper functions (file `tools/populate--stobj-with-prog-bytes.lisp`). We thank Robert Krug for reporting a bug in an earlier version of our model.

References

[1] R. S. Boyer and J S. Moore. Single-threaded Objects in ACL2. In S. Krishnamurthy and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages (PADL)*, volume 2257 of *LNCS*, pages 9–27. Springer-Verlag, 2002.

[2] Warren A. Hunt, Jr. and Matt Kaufmann. An efficient formal model of a large memory. In preparation.

[3] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, Boston, MA, June 2000.

[4] M Kaufmann and J S. Moore. ACL2 home page. See URL `http://www.cs.utexas.edu/users/moore/acl2`.

[5] G. L Steele, Jr. *Common Lisp the Language.* Digital Press, 30 North Avenue, Burlington, MA 01803, 2nd edition, 1990.

7 APPENDIX: Input Files

Here is a summary of the files included below.

- Supporting files:

  `x86-64/constants.lisp`

  `x86-64/misc-events.lisp`

  `x86-64/operations.lisp`

- The x86-64 state, including the memory model:

  `x86-64/x86-state.lisp`

- Read-over-write lemmas

  `x86-64/read-over-write-proofs.lisp` *(proofs are included here)*

  `x86-64/read-over-write.lisp` *includes only results, not proofs*

- Utilities (macros for errors, etc.):

  `x86-64/x86-utils.lisp`

- General support for ISA model (e.g., effective address computation):

  `x86-64/x86-general.lisp`

- The ISA model

  `x86-64/x86.lisp`

- A test

  `tools/script.lisp` *(testbench)*

  `tools/populate-stobj-with-prog-bytes.lisp` *(support)*

  `tools/fib-addr-byte.lisp` *(support)*

8 ========== File x86-64/constants.lisp ==========

```
; constants.lisp                            Warren A. Hunt ,Jr.

; X86 specific constants.  Where possible, these are meant to exactly
; follow the numbers used by X86 binary representations.

(in-package "ACL2")

(include-book "tools/bstar"    :dir :system)
(include-book "misc-events")
(include-book "operations")


(defconst *b-size*   8)  ; Byte
(defconst *w-size*  16)  ; Word
(defconst *d-size*  32)  ; Double
(defconst *q-size*  64)  ; Quad

(defconst *m-size*  64)  ; Machine size (!! change?)

(defconst *default-mem-value*

; If we change this default memory value, we also need to change the :initially
; value in the mem-array field of our x86-64 stobj.

  0)

;  Memory size

(defconst *mem-size-in-bytes*  (expt 2 48))
(defconst *mem-size-in-words*  (floor *mem-size-in-bytes* 2))
(defconst *mem-size-in-dwords* (floor *mem-size-in-bytes* 4))
(defconst *mem-size-in-qwords* (floor *mem-size-in-bytes* 8))

; The following comment also appears in x86-state.lisp; keep these in sync.

; The memory is laid out in a flat array, to be viewed as back-to-back 2MB
; pages.  Here is an example of how that works.  Suppose we have:

;   mem-table[#x7654321] = 0_[18-zeros]
;   mem-table[#x36789ab] = 1_[18-zeros]
;   mem-table[#x2111111] = 2_[18-zeros]

; Then mem-array starts as follows.  Of course, 2^18 qwords = 2^21 bytes, i.e.,
; 2 MB.

;   2^18 qwords corresponding to qword addresses with top 27 bits = #x7654321
;   2^18 qwords corresponding to qword addresses with top 27 bits = #x36789ab
;   2^18 qwords corresponding to qword addresses with top 27 bits = #x2111111
; All additional values in mem-table are the initial value of 1, which means
; "page is not present".

(defconst *initial-mem-array-pages* 100) ; arbitrary
(defconst *initial-mem-array-length* (* *initial-mem-array-pages* (expt 2 18)))
(defconst *2mb-qword-page-size* (floor (expt 2 21) 8)) ; (expt 2 18)
(defconst *mem-table-size*      (floor *mem-size-in-qwords*
                                       *2mb-qword-page-size*)) ; (expt 2 27)
(defconst *mem-array-resize-factor* 3) ; somewhat arbitrary

; Some "useful" constants.  We define these because the ACL2
; definition mechanism does not evaluate and "fold" constants.


(defconst *2^0*       (expt 2  0))
(defconst *2^1*       (expt 2  1))
(defconst *2^2*       (expt 2  2))
(defconst *2^3*       (expt 2  3))
(defconst *2^4*       (expt 2  4))
(defconst *2^5*       (expt 2  5))
(defconst *2^6*       (expt 2  6))
(defconst *2^7*       (expt 2  7))
(defconst *2^8*       (expt 2  8))
(defconst *2^16*      (expt 2 16))
(defconst *2^16-1*    (- *2^16* 1))
(defconst *2^18*      (expt 2 18))
(defconst *2^21*      (expt 2 21))
(defconst *2^24*      (expt 2 24))
(defconst *2^27*      (expt 2 27))
(defconst *2^30*      (expt 2 30))
(defconst *2^32*      (expt 2 32))
(defconst *2^45*      (expt 2 45))
(defconst *2^48*      (expt 2 48))
(defconst *2^48-1*    (- *2^48* 1))
(defconst *2^48-16*   (- *2^48* 16))
(defconst *2^64*      (expt 2 64))
(defconst *2^64-1*    (- *2^64* 1))
(defconst *2^128*     (expt 2 128))

(defconst *2^32-1*    (- *2^32*  1))
(defconst *2^32-2*    (- *2^32*  2))
(defconst *2^32-3*    (- *2^32*  3))
(defconst *2^32-4*    (- *2^32*  4))
(defconst *2^32-5*    (- *2^32*  5))
(defconst *2^32-6*    (- *2^32*  6))
(defconst *2^32-7*    (- *2^32*  7))
(defconst *2^32-8*    (- *2^32*  8))
(defconst *2^32-9*    (- *2^32*  9))
(defconst *2^32-10*   (- *2^32* 10))
(defconst *2^32-11*   (- *2^32* 11))
(defconst *2^32-12*   (- *2^32* 12))
```

```
(defconst *2^32-13*   (- *2^32* 13))
(defconst *2^32-14*   (- *2^32* 14))
(defconst *2^32-15*   (- *2^32* 15))
(defconst *2^32-16*   (- *2^32* 16))

; X86-specific information.

; These numberings are specific to what is used by the X86 processor
; family.

(defconst *x86-64-reg-numbers*
  '((:eax . 0)   (%eax . 0)
    (:ecx . 1)   (%ecx . 1)
    (:edx . 2)   (%edx . 2)
    (:ebx . 3)   (%ebx . 3)
    (:esp . 4)   (%esp . 4)
    (:ebp . 5)   (%ebp . 5)
    (:esi . 6)   (%esi . 6)
    (:edi . 7)   (%edi . 7)

    (:rax . 0)   (%rax . 0)
    (:rcx . 1)   (%rcx . 1)
    (:rdx . 2)   (%rdx . 2)
    (:rbx . 3)   (%rbx . 3)
    (:rsp . 4)   (%rsp . 4)
    (:rbp . 5)   (%rbp . 5)
    (:rsi . 6)   (%rsi . 6)
    (:rdi . 7)   (%rdi . 7)

    (:r00 . 0)   (%r00 . 0)
    (:r01 . 1)   (%r01 . 1)
    (:r02 . 2)   (%r02 . 2)
    (:r03 . 3)   (%r03 . 3)
    (:r04 . 4)   (%r04 . 4)
    (:r05 . 5)   (%r05 . 5)
    (:r06 . 6)   (%r06 . 6)
    (:r07 . 7)   (%r07 . 7)

    (:r08 . 8)   (%r08 . 8)
    (:r09 . 9)   (%r09 . 9)
    (:r10 . 10)  (%r10 . 10)
    (:r11 . 11)  (%r11 . 11)
    (:r12 . 12)  (%r12 . 12)
    (:r13 . 13)  (%r13 . 13)
    (:r14 . 14)  (%r14 . 14)
    (:r15 . 15)  (%r15 . 15)
    ))

(defun x86-rton (name)
  (declare (xargs :guard (symbolp name)))
  (cdr (assoc name *x86-64-reg-numbers*)))
```

```
(defmacro rtn (name)
  `(x86-rton ,name))

; X86-specific registers and their sub-fields

(defconst *x86-eflags*
  '((:cf    0  1)  ; Carry Flag
    (1      1  1)  ; 1
    (:pf    2  1)  ; Parity Flag
    (0      3  1)  ; 0
    (:af    4  1)  ; Auxiliary-carry Flag
    (0      5  1)  ; 0
    (:zf    6  1)  ; Zero Flag
    (:sf    7  1)  ; Sign Flag
    (:tf    8  1)  ; Trap Flag
    (:if    9  1)  ; Interrupt-enable Flag
    (:df   10  1)  ; Direction Flag
    (:of   11  1)  ; Overflow Flag
    (:iopl 12  2)  ; I/O Privilege Level
    (:nt   14  1)  ; Nested Task
    (0     15  1)  ; 0
    (:rf   16  1)  ; Resume Flag
    (:vm   17  1)  ; Virtual-8086 Mode
    (:ac   18  1)  ; Alignment Check
    ;; Pentium and follow-on processors include additional flags
    (:vif  19  1)  ; Virtual Interrupt Flag
    (:vip  20  1)  ; Virtual Interrupt Pending
    (:id   21  1)  ; ID flag
    (0     22 10)  ; 0
    ))

(defthm x86-eflags-table-ok
  (reg-info-alistp *x86-eflags* 0 *m-size*)
  :rule-classes nil)

(defmacro x86-eflags (flg eflags)
  (x86-reg-slice flg eflags *x86-eflags*))

(defmacro x86-update-eflags (flg val eflags)
  (x86-update-reg-slice flg val eflags *x86-eflags*))

; The next events should perhaps be generalized beyond just eflags.

(defun flg-to-nat (flg)
  (declare (xargs :guard
                  (and (symbolp flg)
                       (mv-let (pos width)
                               (x86-flg-field-pos-width flg *x86-eflags*)
                               (declare (ignore pos))
                               (eql width 1)))))
  (mv-let (pos width)
          (x86-flg-field-pos-width flg *x86-eflags*)
          (cond ((eql width 1)
                 (ash 1 pos))
                (t (prog2$ (er hard? 'flg-to-nat
                               "Bad flag key, ~x0."
                               flg)
                           0)))))

(defun flg-to-nat* (flg-list)
; Flg-list should be a list of flag keywords, each bound in *x86-eflags* to a
; field of width 1 (else we'll get an error).
  (declare (xargs :guard (symbol-listp flg-list)))
  (cond ((endp flg-list) 0)
        (t (logior (ec-call (flg-to-nat (car flg-list)))
                   (flg-to-nat* (cdr flg-list))))))

(defun flg-val (val pos)
  (declare (xargs :guard (and (booleanp val)
                              (natp pos)
                              (< pos 64))))
  (if val (ash 1 pos) 0))

(encapsulate
 ()
 (local (include-book "arithmetic/top" :dir :system))

 (defthm flg-val<2^64
   (implies (and (natp pos)
                 (< pos 64))
            (<= (flg-val val pos)
                *2^64-1*))
   :rule-classes :linear))

(defthm natp-flg-val
   (implies (force (natp pos))
            (natp (flg-val val pos)))
   :rule-classes :type-prescription)

(defun flg-val-form (flg val)
  (declare (xargs :guard (keywordp flg)))
  (mv-let (pos width)
          (x86-flg-field-pos-width flg *x86-eflags*)
          (declare (ignore width)) ; checked in above call
          `(mbe :logic
                (flg-val ,val ,pos)
                :exec
                (if ,val ,(ash 1 pos) 0))))

(defun flg-val-forms (x)
  (declare (xargs :guard (keyword-value-listp x)))
  (cond ((endp x) nil)
        (t (cons (let ((flg (car x))
                       (val (cadr x)))
                   (flg-val-form flg val))
                 (flg-val-forms (cddr x))))))

(defmacro !arith-flags (flags &rest flg-kwd-value-list
                              &key cf pf af sf of zf)
  (declare (ignore cf pf af sf of zf))
  (let ((mask (logand *2^64-1*
                      (lognot (flg-to-nat* (evens flg-kwd-value-list))))))
    `(logior (logand ,mask ,flags)
             ,@(flg-val-forms flg-kwd-value-list))))

; !! Eliminate or change segment stuff?

(defconst *x86-segment-selector*
  '((:rpl   0  2)  ; Requestor Privilege Level (RPL)
    (:ti    2  1)  ; Table Indicator (0 = GDT, 1 = LDT)
    (:index 3 13)  ; Index of descriptor in GDT or LDT
    ))

(defthm x86-segment-selector-ok
  (reg-info-alistp *x86-segment-selector* 0 *w-size*)
  :rule-classes nil)

(defmacro x86-segment-selector (flg eflags)
  (x86-reg-slice flg eflags *x86-segment-selector*))

(defmacro x86-update-segment-selector (flg val eflags)
  (x86-update-reg-slice flg val eflags *x86-segment-selector*))


(defconst *gdtr-offset* 0)
(defconst *idtr-offset* 1)

(defconst *x86-system-table-register-selector*
  '((:limit 0 16)
    (:base 16 32)))

;; Fix constant (expt 2 48)!!!
(defthm x86-system-table-register-selector-ok
  (reg-info-alistp *x86-system-table-register-selector* 0 (expt 2 48))
  :rule-classes nil)

(defmacro x86-system-table-register-selector (flg eflags)
  (x86-reg-slice flg eflags
                 *x86-system-table-register-selector*))

(defmacro x86-update-system-table-register-selector (flg val eflags)
  (x86-update-reg-slice flg val eflags
                       *x86-system-table-register-selector*))


(defconst *x86-seg-descriptor-0-fields*
  '((:base-15_0        0 16) ; Segment Base Address (bits 15..0)
    (:seg-limit-15_0  16 16) ; Segment Limit (bits 15..0)
    ))

(defthm x86-seg-descriptor-0-fields-ok
  (reg-info-alistp *x86-seg-descriptor-0-fields* 0 *m-size*)
  :rule-classes nil)

(defmacro x86-seg-descriptor-field-0 (flg seg-field)
  (x86-reg-slice flg seg-field *x86-seg-descriptor-0-fields*))

(defmacro x86-update-seg-descriptor-field-0 (flg val seg-field)
  (x86-update-reg-slice flg val seg-field *x86-seg-descriptor-0-fields*))


(defconst *x86-seg-descriptor-1-fields*
  '((:base-23_16       0  8) ; Segment Base Address (bits 23..16)
    (:accessed         8  1) ; Accessed
    (:w/r              9  1) ; read-Write (for data); execute-Read (for code)
    (:e/c             10  1) ; Expand-down (for data); Conforming (for code)
    (:data/code       11  1) ; Data (0) segment or Code (1) segment
    (:system          12  1) ; System (0) segment or code/data (1) segment, effects
                             ; meaning of 4 bits above.
    (:dpl             13  2) ; Descriptor Privilege Level (0 -- OS, ... 3 -- user)
    (:present         15  1) ; descriptor Present
    (:seg-limit-19_16 16  4) ; Segment Limit (bits 19..16)
    (:avaiable        20  1) ; Available
    (0                21  1) ; 0
    (:d/b             22  1) ; Default-operation size (1 for 32-bit, 0 for 16-bit)
    (:granularity     23  1) ; Granularity
    (:base-31_24      24  8) ; Segment Base Address (bits 31..24)
    ))

(defthm x86-seg-descriptor-1-fields-ok
  (reg-info-alistp *x86-seg-descriptor-1-fields* 0 *m-size*)
  :rule-classes nil)

(defmacro x86-seg-descriptor-field-1 (flg seg-field)
  (x86-reg-slice flg seg-field *x86-seg-descriptor-1-fields*))

(defmacro x86-update-seg-descriptor-field-1 (flg val seg-field)
  (x86-update-reg-slice flg val seg-field *x86-seg-descriptor-1-fields*))


(defconst *x86-cr0*
  '((:pe    0  1)  ; Protection Enable
    (:mp    1  1)  ; Monitor coProcessor
    (:em    2  1)  ; Emulation Mode (for coprocessors)
    (:ts    3  1)  ; Task Switched
    (:et    4  1)  ; Extension Type
    (:ne    5  1)  ; Numeric Error
    (0      6 10)  ; 0
    (:wp   16  1)  ; Write Protect
    (0     17  1)  ; 0
    (:am   18  1)  ; Alignment Mask
    (0     19 10)  ; 0
    (:nw   29  1)  ; Not Write-through
    (:cd   30  1)  ; Cache Disable
    (:pg   31  1)  ; PaGing enable
    ))

(defthm x86-cr0-table-ok
  (reg-info-alistp *x86-cr0* 0 *m-size*)
  :rule-classes nil)

(defmacro x86-cr0 (flg cr0)
  (x86-reg-slice flg cr0 *x86-cr0*))

(defmacro x86-update-cr0 (flg val cr0)
  (x86-update-reg-slice flg val cr0 *x86-cr0*))


(defconst *x86-cr3*
  '((0      0  3)  ; 0
    (:pwt   3  1)  ; Page-Llevel Writes Tranparent
    (:pcd   4  1)  ; Page-level Cache Disable
    (0      5  7)  ; 0
    (:pdb  12 20)  ; Page Directory Base
    ))

(defthm x86-cr0-table-ok
  (reg-info-alistp *x86-cr0* 0 *m-size*)
  :rule-classes nil)

(defmacro x86-cr3 (flg cr3)
  (x86-reg-slice flg cr3 *x86-cr3*))

(defmacro x86-update-cr3 (flg val cr3)
  (x86-update-reg-slice flg val cr3 *x86-cr3*))


(defconst *x86-virtual-addr*
  '((:poffset  0 12) ; Offset into page
    (:pte     12 10) ; Page-table index
    (:pde     22 10) ; Page-directory index
    ))

(defthm x86-virtual-address-table-ok
  (reg-info-alistp *x86-virtual-addr* 0 *m-size*)
  :rule-classes nil)

(defmacro x86-virtual-addr-slice (flg addr)
  (x86-reg-slice flg addr *x86-virtual-addr*))

(defmacro x86-update-virtual-addr-slice (flg addr val)
  (x86-update-reg-slice flg val addr *x86-virtual-addr*))


(defconst *x86-page-directory-entry*
  '((:present        0  1) ; Present
    (:read/write      1  1) ; Read/Write
    (:user/supervisor 2  1) ; User/Supervisor
    (:write-through   3  1) ; Write-through
    (:cache-disabled  4  1) ; Cache disabled
    (:accessed        5  1) ; Accessed
    (0                6  1) ; Reserved (set to 0)
    (:page-size       7  1) ; Page-size (0 indicated 4K bytes)
    (:global-table    8  1) ; Global table (ingored)
    (:available       9  3) ; Available for OS programmer
    (:pg-tb-bs-addr  12 20) ; Page-table-base address
    ))

(defthm x86-page-directory-entry-ok
  (reg-info-alistp *x86-page-directory-entry* 0 *m-size*)
  :rule-classes nil)

(defmacro x86-page-directory-slice (flg addr)
  (x86-reg-slice flg addr *x86-page-directory-entry*))

(defmacro x86-update-page-directory-slice (flg addr val)
  (x86-update-reg-slice flg val addr *x86-page-directory-entry*))


(defconst *x86-page-table-entry*
  '((:present        0  1) ; Present
    (:read/write      1  1) ; Read/Write
    (:user/supervisor 2  1) ; User/Supervisor
    (:write-through   3  1) ; Write-through
    (:cache-disabled  4  1) ; Cache disabled
    (:accessed        5  1) ; Accessed
    (:dirty           6  1) ; Dirty
    (0                7  1) ; Reserved, set to 0
    (:global-table    8  1) ; Global table (ingored)
    (:available       9  3) ; Available for OS programmer
    (:pg-base-addr   12 20) ; Base address of page
    ))

(defthm x86-page-table-entry-ok
  (reg-info-alistp *x86-page-directory-entry* 0 *m-size*)
  :rule-classes nil)

(defmacro x86-page-table-slice (flg addr)
  (x86-reg-slice flg addr *x86-page-directory-entry*))

(defmacro x86-update-page-table-slice (flg addr val)
  (x86-update-reg-slice flg val addr *x86-page-directory-entry*))


; ACL2 X86 model-specific information.

; We are trying to write a model of the X86.  Below, we have attempted
; to identify the "top-level" X86 registers.  We are building several
; models on top of this state representation.

; We use an alternate copy of much of this information for our Y86 model.  The
; Y86 is a subset of the X86, and we reuse the X86 representation of the state
; for our Y86 model; however, the Y86 just has three 1-bit, flag (:sf :zf :of)
; registers.  Even so, our Y86 model uses the corresponding three bits from the
; 32-bit X86 EFLAGS register.

; The structure of our 64-bit X86 state is a list (actually array) of values.
; Thus, :eip has address 0.  These positions and addresses are independent of
; the indicies used internally by the X86.

(defconst *m86-64-reg-names*
  ;; 64-bit GP registers, :eax "address" is 0
  ;; For proper order of the first 8 registers, see:
  ;; http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-vol-2a-manua
  `(:rax :rcx :rdx :rbx :rsp :rbp :rsi :rdi ; see
         :r8 :r9 :r10 :r11 :r12 :r13 :r14 :r15))

(defconst *m86-64-segment-reg-names*
  ;; 64-bit segment register, :es "address is 0
  `(:es :cs :ss :ds :fs :gs))

(defconst *m86-64-gdtr-idtr-names*
  ;; 48-bit system-table (GDTR and IDTR) registers
  '(:gdtr :idtr))

(defconst *m86-64-ldtr-tr-names*
  ;; 16-bit system-segment (Task Register and LDTR) registers
  '(:ldtr :tr))

(defconst *m86-64-control-reg-names*
  '(:msw                                     ;; Control status register
    :cr0 :cr1 :cr2 :cr3 :cr4 :cr5 :cr6 :cr7   ;; Control registers
    :cr8 :cr9 :cr10 :cr11 :cr12 :cr13 :cr14 :cr15
    :xcr0))

(defconst *m86-64-debug-reg-names*
  '(:dr0 :dr1 :dr2 :dr3 :dr4 :dr5 :dr6 :dr7)) ;; Debug registers

(defconst *m86-64-fp-reg-names*
  `(:fp0 :fp1 :fp2 :fp3 :fp4 :fp5 :fp6 :fp7   ;; Floating-point, 80-bit registers
    :fp-cntl :fp-status :fp-tag :fp-opcode    ;; 16-bit FP status registers
    :fp-last-inst :fp-last-data))             ;; 48-bit last-instruction registers

(defconst *m86-64-xmm-reg-names*
  `(:mxcsr                                   ;; 64-bit status register
    :xmm0 :xmm1 :xmm2 :xmm3                  ;; 128-bit data registers
    :xmm0 :xmm5 :xmm6 :xmm7))

(defconst *m86-64-model-specific-registers*
  `(;; This is a complex internal X86 state.
    ;; At this point, we do not model the X86 MSRs.
    ))


(defconst *m86-64-reg-names-len*          (len *m86-64-reg-names*))
(defconst *m86-64-segment-reg-names-len*  (len *m86-64-segment-reg-names*))
(defconst *m86-64-gdtr-idtr-names-len*    (len *m86-64-gdtr-idtr-names*))
(defconst *m86-64-ldtr-tr-names-len*      (len *m86-64-ldtr-tr-names*))

(defconst *m86-64-control-reg-names-len*  (len *m86-64-control-reg-names*))
(defconst *m86-64-debug-reg-names-len*    (len *m86-64-debug-reg-names*))
(defconst *m86-fp-reg-names-len*          (len *m86-64-fp-reg-names*))
(defconst *m86-64-xmm-reg-names-len*      (len *m86-64-xmm-reg-names*))


(defun m86-64-reg-pos (name state-names n)
  (declare (xargs :guard (and (eqlablep name) (natp n))))
  (if (atom state-names)
      0
    (if (eql name (car state-names))
        n
      (m86-64-reg-pos name (cdr state-names) (1+ n)))))

(defconst *mr-eax* (m86-64-reg-pos :eax *m86-64-reg-names* 0))
(defconst *mr-ecx* (m86-64-reg-pos :ecx *m86-64-reg-names* 0))
(defconst *mr-edx* (m86-64-reg-pos :edx *m86-64-reg-names* 0))
(defconst *mr-ebx* (m86-64-reg-pos :ebx *m86-64-reg-names* 0))
(defconst *mr-esp* (m86-64-reg-pos :esp *m86-64-reg-names* 0))
(defconst *mr-ebp* (m86-64-reg-pos :ebp *m86-64-reg-names* 0))
(defconst *mr-esi* (m86-64-reg-pos :esi *m86-64-reg-names* 0))
(defconst *mr-edi* (m86-64-reg-pos :edi *m86-64-reg-names* 0))

(defconst *mr-rax* (m86-64-reg-pos :rax *m86-64-reg-names* 0))
(defconst *mr-rcx* (m86-64-reg-pos :rcx *m86-64-reg-names* 0))
(defconst *mr-rdx* (m86-64-reg-pos :rdx *m86-64-reg-names* 0))
(defconst *mr-rbx* (m86-64-reg-pos :rbx *m86-64-reg-names* 0))
(defconst *mr-rsp* (m86-64-reg-pos :rsp *m86-64-reg-names* 0))
```

```
(defconst *mr-rbp* (m86-64-reg-pos :rbp *m86-64-reg-names* 0))
(defconst *mr-rsi* (m86-64-reg-pos :rsi *m86-64-reg-names* 0))
(defconst *mr-rdi* (m86-64-reg-pos :rdi *m86-64-reg-names* 0))

(defconst *mr-r00* (m86-64-reg-pos :r00 *m86-64-reg-names* 0))
(defconst *mr-r01* (m86-64-reg-pos :r01 *m86-64-reg-names* 0))
(defconst *mr-r02* (m86-64-reg-pos :r02 *m86-64-reg-names* 0))
(defconst *mr-r03* (m86-64-reg-pos :r03 *m86-64-reg-names* 0))
(defconst *mr-r04* (m86-64-reg-pos :r04 *m86-64-reg-names* 0))
(defconst *mr-r05* (m86-64-reg-pos :r05 *m86-64-reg-names* 0))
(defconst *mr-r06* (m86-64-reg-pos :r06 *m86-64-reg-names* 0))
(defconst *mr-r07* (m86-64-reg-pos :r07 *m86-64-reg-names* 0))

(defconst *mr-r08* (m86-64-reg-pos :r08 *m86-64-reg-names* 0))
(defconst *mr-r09* (m86-64-reg-pos :r09 *m86-64-reg-names* 0))
(defconst *mr-r10* (m86-64-reg-pos :r10 *m86-64-reg-names* 0))
(defconst *mr-r11* (m86-64-reg-pos :r11 *m86-64-reg-names* 0))
(defconst *mr-r12* (m86-64-reg-pos :r12 *m86-64-reg-names* 0))
(defconst *mr-r13* (m86-64-reg-pos :r13 *m86-64-reg-names* 0))
(defconst *mr-r14* (m86-64-reg-pos :r14 *m86-64-reg-names* 0))
(defconst *mr-r15* (m86-64-reg-pos :r15 *m86-64-reg-names* 0))



(defun m86-reg-pos1 (reg reg-names n)
  (declare (xargs :guard (and (symbolp reg)
                              (symbol-listp reg-names)
                              (natp n))))
  (if (atom reg-names)
      (or (cw "m86-reg-pos1:  Name not found:  ~p0.~%" reg) 0)
    (if (eq (car reg-names) reg)
        n
      (m86-reg-pos1 reg (cdr reg-names) (1+ n)))))

(defthm natp-m86-reg-pos1
  (implies (natp n)
           (and (integerp (m86-reg-pos1 reg reg-names n))
                (<= 0 (m86-reg-pos1 reg reg-names n))))
  :rule-classes :type-prescription)

(defun m86-reg-pos (reg reg-names)
  (declare (xargs :guard (and (symbolp reg)
                              (symbol-listp reg-names))))
  (m86-reg-pos1 reg reg-names 0))

(defthm natp-m86-reg-pos
  (and (integerp (m86-reg-pos reg reg-names))
       (<= 0 (m86-reg-pos reg reg-names)))
  :rule-classes :type-prescription)

(in-theory (disable m86-reg-pos))
```

```
(defconst *cr0* (m86-reg-pos :cr0 *m86-64-control-reg-names*))
(defconst *cr1* (m86-reg-pos :cr1 *m86-64-control-reg-names*))
(defconst *cr2* (m86-reg-pos :cr2 *m86-64-control-reg-names*))
(defconst *cr3* (m86-reg-pos :cr3 *m86-64-control-reg-names*))
(defconst *cr4* (m86-reg-pos :cr4 *m86-64-control-reg-names*))

(defconst *x86-32-bit-lap-nops*
 ;; "Allegedly, many implementations recognize these instructions and
 ;; execute them very quickly."  This information below actually appears
 ;; in the CCL X86 compiler, but similar information is available in the
 ;; Intel X86 documentation.
 `(()
   (#x90)                                   ; nop
   (#x89 #xf6)                              ; movl %esi,%esi
   (#x8d #x76 #x00)                         ; leal 0(%esi),%esi
   (#x8d #x74 #x26 #x00)                    ; leal 0(%esi,1),%esi
   (#x90 #x8d #x74 #x26 #x00)               ; nop ; leal 0(%esi,1),%esi
   (#x8d #xb6 #x00 #x00 #x00 #x00)          ; leal 0L(%esi),%esi
   (#x8d #xb4 #x26 #x00 #x00 #x00 #x00)     ; leal 0L(%esi,1),%esi
   (#x90 #x8d #xb4 #x26 #x00 #x00 #x00 #x00) ; nop, and line above
 ))

(defconst *onebyte-has-modrm-lst*
 '(
   #|       0 1 2 3 4 5 6 7 8 9 a b c d e f        |#
   #|       -------------------------------        |#
   #| 00 |# 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0  #| 00 |#
   #| 10 |# 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0  #| 10 |#
   #| 20 |# 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0  #| 20 |#
   #| 30 |# 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0  #| 30 |#
   #| 40 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| 40 |#
   #| 50 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| 50 |#
   #| 60 |# 0 0 1 1 0 0 0 0 0 1 0 1 0 0 0 0  #| 60 |#
   #| 70 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| 70 |#
   #| 80 |# 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  #| 80 |#
   #| 90 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| 90 |#
   #| a0 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| a0 |#
   #| b0 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| b0 |#
   #| c0 |# 1 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0  #| c0 |#
   #| d0 |# 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1  #| d0 |#
   #| e0 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| e0 |#
   #| f0 |# 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1  #| f0 |#
   #|       -------------------------------        |#
   #|       0 1 2 3 4 5 6 7 8 9 a b c d e f        |#
   ))

(defconst *onebyte-has-modrm-ar*
  (list-to-array 'onebyte-has-modrm
                 (ints-to-booleans *onebyte-has-modrm-lst*)))
```

```
(defconst *twobyte-has-modrm*
 '(
   #|       0 1 2 3 4 5 6 7 8 9 a b c d e f        |#
   #|       -------------------------------        |#
   #| 00 |# 1 1 1 1 0 0 0 0 0 0 0 0 0 1 0 1  #| 0f |#
   #| 10 |# 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0  #| 1f |#
   #| 20 |# 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1  #| 2f |#
   #| 30 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| 3f |#
   #| 40 |# 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  #| 4f |#
   #| 50 |# 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  #| 5f |#
   #| 60 |# 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  #| 6f |#
   #| 70 |# 1 1 1 1 1 1 1 0 0 0 0 0 1 1 1 1  #| 7f |#
   #| 80 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| 8f |#
   #| 90 |# 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  #| 9f |#
   #| a0 |# 0 0 0 1 1 1 1 1 0 0 0 1 1 1 1 1  #| af |#
   #| b0 |# 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1  #| bf |#
   #| c0 |# 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0  #| cf |#
   #| d0 |# 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  #| df |#
   #| e0 |# 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  #| ef |#
   #| f0 |# 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0  #| ff |#
   #|       -------------------------------        |#
   #|       0 1 2 3 4 5 6 7 8 9 a b c d e f        |#
   ))

(defconst *twobyte-uses-sse-prefix*
 '(
   #|       0 1 2 3 4 5 6 7 8 9 a b c d e f        |#
   #|       -------------------------------        |#
   #| 00 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| 0f |#
   #| 10 |# 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0  #| 1f |#
   #| 20 |# 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0  #| 2f |#
   #| 30 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| 3f |#
   #| 40 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| 4f |#
   #| 50 |# 0 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1  #| 5f |#
   #| 60 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1  #| 6f |#
   #| 70 |# 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1  #| 7f |#
   #| 80 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| 8f |#
   #| 90 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| 9f |#
   #| a0 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| af |#
   #| b0 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| bf |#
   #| c0 |# 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0  #| cf |#
   #| d0 |# 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0  #| df |#
   #| e0 |# 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0  #| ef |#
   #| f0 |# 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0  #| ff |#
   #|       -------------------------------        |#
   #|       0 1 2 3 4 5 6 7 8 9 a b c d e f        |#
   ))


; NOTE!  Some of the text below was "lifted" from Intel's documentation.
; Instruction Prefixes

; Instruction prefixes are divided into four groups, each with a set
; of allowable prefix codes. For each instruction, it is only useful
; to include up to one prefix code from each of the four groups
; (Groups 1, 2, 3, 4). Groups 1 through 4 may be placed in any order
; relative to each other.

; Group 1 - Lock and repeat prefixes:
; * LOCK prefix is encoded using F0H
; * REPNE/REPNZ prefix is encoded using F2H. Repeat-Not-Zero prefix
;   applies only to string and input/output instructions. (F2H is also
;   used as a mandatory prefix for some instructions)
; * REP or REPE/REPZ is encoded using F3H. Repeat prefix applies only
;   to string and input/output instructions.(F3H is also used as a
;   mandatory prefix for some instructions)

; Group 2 - Segment override prefixes:
; * 2EH - CS segment override (use with any branch instruction is reserved)
; * 36H - SS segment override (use with any branch instruction is reserved)
; * 3EH - DS segment override (use with any branch instruction is reserved)
; * 26H - ES segment override (use with any branch instruction is reserved)
; * 64H - FS segment override (use with any branch instruction is reserved)
; * 65H - GS segment override (use with any branch instruction is reserved)

; Group 2 - Branch hints:
; * 2EH - Branch not taken (used only with Jcc instructions)
; * 3EH - Branch taken (used only with Jcc instructions)

; Group 3
; * Operand-size override prefix is encoded using 66H (66H is also used as a
;   mandatory prefix for some instructions).

; Group 4
; * 67H - Address-size override prefix

; The LOCK prefix (F0H) forces an operation that ensures exclusive use
; of shared memory in a multiprocessor environment.

; Repeat prefixes (F2H, F3H) cause an instruction to be repeated for
; each element of a string. Use these prefixes only with string and I/O
; instructions (MOVS, CMPS, SCAS, LODS, STOS, INS, and OUTS). Use of
; repeat prefixes and/or undefined opcodes with other Intel 64 or IA-32
; instructions is reserved; such use may cause unpredictable behavior.

; Some instructions may use F2H, F3H as a mandatory prefix to express
; distinct functionality.  A mandatory prefix generally should be placed
; after other optional prefixes (exception to this is discussed in
; Section 2.2.1, "REX Prefixes")

; Branch hint prefixes (2EH, 3EH) allow a program to give a hint to the
; processor about the most likely code path for a branch. Use these
; prefixes only with conditional branch instructions (Jcc). Other use of
; branch hint prefixes and/or other undefined opcodes with Intel 64 or
; IA-32 instructions is reserved; such use may cause unpredictable
; behavior.

; The operand-size override prefix allows a program to switch between
; 16- and 32-bit operand sizes. Either size can be the default; use of
; the prefix selects the non-default size.

; Some SSE2/SSE3/SSSE3/SSE4 instructions and instructions using a
; three-byte sequence of primary opcode bytes may use 66H as a mandatory
; prefix to express distinct functionality. A mandatory prefix generally
; should be placed after other optional prefixes (exception to this is
; discussed in Section "REX Prefixes"). Other use of the 66H prefix is
; reserved; such use may cause unpredictable behavior.  The address-size
; override prefix (67H) allows programs to switch between 16- and 32-bit
; addressing. Either size can be the default; the prefix selects the
; non-default size. Using this prefix and/or other undefined opcodes
; when operands for the instruction do not reside in memory is reserved;
; such use may cause unpredictable behavior.

(defconst *in0*
 ;; First byte of instruction, after prefix byte(s).
 '(
   ;; EX override prefix
   (#x26 :prefix :cs-override :group 2
         ;; Hint when used with a branch instruction prefix.
         :branch :not-taken)

   ;; CS override prefix
   (#x2E :prefix :cs-override :group 2)

   ;; SS override prefix
   (#x36 :prefix :ss-override :group 2)

   ;; DS override prefix
   (#x3E :prefix :ds-override :group 2
         ;; Hint when used as a branch instruction prefix.
         :branch :taken)

   ;; FS override prefix
   (#x64 :prefix :fs-override :group 2)

   ;; GS override prefix
   (#x65 :prefix :gs-override :group 2)

   ;; Operand-size override prefix
   (#x66 :prefix :operand-size-override :group 3)

   ;; Address-size override prefix
   (#x67 :prefix :address-size-override :group 4)

   ;; MOV: Move r8 to r/m8
   (#x88 :ModR/M t)

   ;; MOV: Move r16 to r/m16 or r32 to r/m32
   (#x89 :ModR/M t)

   ;; MOV: Move r/m8 to r8
   (#x8A :ModR/M t)

   ;; MOV: Move r/m16 to r16 or r/m32 to r32
   (#x8B :ModR/M t)

   ;; NOP
   (#x90 :length 1
         :op
         ((:eip (+32 :eip 1))))

   ;; JMP with 8-bit displacement, relative to byte address just past
   ;; the end of this instruction.
   (#xEB :length 2
         :op
         ((disp (s8_to_s32 (+32 :eip 1)))
          (:eip (+32 (+32 :eip 2) disp))))

   ;; LOCK prefix forces an operation that ensure exclusive use of
   ;; shared memory in a multiprocessor environment.
   (#xF0 :prefix :lock :group 1)

   ;; REPNE/REPNZ for string and I/O instructions.  Causes an
   ;; instruction to be repeated for each element of a string (MOVS,
   ;; CMPS, SCAS, LODS, STOS, INS, and OUTS).
   (#xF2 :prefix :repnz :group 1)

   ;; REP or REPE/REPZ for string and I/O instructions (see REPNZ).
   (#xF3 :prefix :rep :group 1)
   ))
```

9 ========== File x86-64/misc-events.lisp ==========

```
; misc-events.lisp                             Warren A. Hunt, Jr.

; WAH,Jr.

(in-package "ACL2")

; To monitor a rewrite rule <rewrite-rule>:
; :brr t
; :monitor (:rewrite <rewrite-rule>) t


; Read about TYPE-PRESCRIPTION rules.

; (set-gag-mode nil)  ; To get all output.


; Some miscellaneous definitions.

(defmacro ! (x y)
  (declare (xargs :guard (symbolp x)))
  `(assign ,x ,y))

(defmacro !! (variable new-value)
  ;; Assign without printing the result.
  (declare (xargs :guard t))
  `(mv-let
    (erp result state)
    (assign ,variable ,new-value)
    (declare (ignore result))
    (value (if erp 'Error! ',variable))))


; Some help with some arithmetic issues.

(local (include-book "arithmetic-5/top" :dir :system))
(local (include-book "rtl/rel8/lib/top" :dir :system))

(defmacro with-arithmetic-help-5 (&rest forms)
  `(encapsulate
    ()
    (local (include-book "arithmetic-5/top" :dir :system))
    (set-default-hints '((nonlinearp-default-hint
                          stable-under-simplificationp
                          hist
                          pspv)))
    ,@forms))

; Functions repeatedly used in processor definitions:
;   LOGAND, LOGIOR, LOGXOR, LOGNOT, and ASH.
; When using such function, I want to know that the result is a
; bounded, natural number.  In each of the LOGNOT, this isn't true,
; because any positive number become negative; thus, any use of LOGNOT
; will likely be wrapped inside a LOGAND.

; ASH rules

(with-arithmetic-help-5
 (defthm ash-negative-shift-makes-input-smaller

; The syntaxp hypothesis below probably speeds up proofs significantly towards
; the end of y86/read-over-write.lisp.

   (implies (and (syntaxp (and (quotep shift)
                               (integerp (cadr shift))
                               (< (cadr shift) 0)))
                 (integerp x)
                 (< 0 x)
                 (integerp shift)
                 (< shift 0))
            (< (ash x shift) x))
   :rule-classes :linear))

; LOGAND rules.

(with-arithmetic-help-5
 (defthm logand-less-than-or-equal
   (implies (natp x) ; avoid forcing, else (defstep mov ...) fails
            (and (<= (binary-logand x y) x)
                 (<= (binary-logand y x) x)))
   :hints (("Goal" :cases ((equal x 0))))
   :rule-classes :linear))

(with-arithmetic-help-5
 (defthm logand-greater-or-equal-to-zero
   ;; (NATP (LOGAND x y))
   (implies (or (natp x) (natp y))
            (and (integerp (binary-logand x y))
                 (<= 0 (binary-logand x y))
                 ;; (integerp (binary-logand y x))
                 ;; (<= 0 (binary-logand y x))
                 ))
   :hints (("Goal" :cases ((equal x 0))))
   :rule-classes :type-prescription))

; LOGIOR rules.

(with-arithmetic-help-5
 (defthm logior-greater-or-equal-to-zero
   ;; (NATP (LOGIOR x y))
   (implies (and (natp x) (natp y))
            (and (integerp (logior x y))
                 (<= 0 (logior x y))
                 ;; (<= 0 (logior y x))
                 ))
   :rule-classes
   (:type-prescription
    (:linear :corollary ; originally needed for x86-32p-wm16 in x86-state.lisp
             (implies (and (natp x) (natp y))
                      (<= 0 (logior x y)))))))

(defthm logior-less-than-2^n
  (implies (and (natp x) (natp y)
                (< x (expt 2 n))
                (< y (expt 2 n)))
           (< (logior x y) (expt 2 n)))
  :hints (("Goal"
           :use logior-bvecp
           :in-theory (e/d (bvecp) (logior-bvecp))))
  :rule-classes :linear)


; LOGXOR rules.

(defthm logxor-greater-or-equal-to-zero
  ;; (NATP (LOGXOR x y))
  (implies (and (natp x) (natp y))
           (and (integerp (logxor x y))
                (<= 0 (logxor x y))
                ;; (integerp (logxor y x))
                ;; (<= 0 (logxor y x))
                ))
  :rule-classes :type-prescription)

;; This next rule would be a weird rewrite rule because of the (EXPT
;; 2 N) in the conclusion.  As a linear rule, then entire conclusion
;; doesn't need to match.

(defthm logxor-<=-expt-2-to-n
  (implies (and (natp x) (natp y)
                (< x (expt 2 n))
                (< y (expt 2 n)))
           (< (logxor x y) (expt 2 n)))
  :hints (("Goal"
           :use logxor-bvecp
           :in-theory (e/d (bvecp) (logxor-bvecp))))
  :rule-classes :linear)

; From rtl library:
(defthm integerp-mod
  (implies (and (integerp m)
                (integerp n))
           (integerp (mod m n)))
  :rule-classes (:rewrite :type-prescription))


(defun gl-int (start by count)
  (declare (xargs :guard (and (natp start)
                              (natp by)
                              (natp count))))
  (if (zp count)
      nil
    (cons start
          (gl-int (+ by start) by (1- count)))))


; Below are some functions and macros to assist with defining
; bit-field information.

(defun reg-info-alistp (alst position max-size)
  ;; Recongnizer for register information
  (declare (xargs :guard (and (natp position)
                              (natp max-size))))
  (if (atom alst)
      (null alst)
    (let ((entry (car alst)))
      (and (consp entry)
           (consp (cdr entry))
           (consp (cddr entry))
           (null (cdddr entry))
           (let ((key (car entry))
                 (pos (cadr entry))
                 (width (caddr entry)))
             (and (or (keywordp key)
                      (and (natp key)
                           (or (= key 0)
                               (= key 1))))
                  (natp pos)
                  (natp width)
                  (= position pos)
                  (<= (+ pos width) max-size)
                  (reg-info-alistp (cdr alst)
                                   (+ pos width)
                                   max-size)))))))

(defun x86-flg-field-pos-width (flg reg-info)
  (declare (xargs :guard (symbolp flg)))
  (if (atom reg-info)
      (mv 0 (or (cw "x86-flg-field-pos-width:  Unknown flag:   ~p0.~%" flg) 0))
    (let ((entry (car reg-info)))
      (if (not (and (consp entry)
                    (consp (cdr entry))
```

```
                   (consp (cddr entry))
                   (null (cdddr entry))
                   (or (symbolp (car entry))
                       (natp    (car entry)))
                   (natp (cadr entry))
                   (natp (caddr entry))))
         (mv 0 (or (cw "x86-flg-field-pos-width:  Entry malformed:   ~p0.~%" entry) 0))
       (let ((name (car entry))
             (pos  (cadr entry))
             (width (caddr entry)))
         (if (eq name flg)
             (mv pos width)
           (x86-flg-field-pos-width flg (cdr reg-info))))))))

(defun x86-reg-slice (flg reg reg-info)
  (declare (xargs :guard (symbolp flg)))
  (mv-let (pos size)
    (x86-flg-field-pos-width flg reg-info)
    (let ((mask (1- (expt 2 size))))
      `(logand ,mask
               (ash ,reg (- ,pos))))))



(defun x86-reg-field (flg reg reg-info)
  (declare (xargs :guard (symbolp flg)))
  (mv-let (pos size)
    (x86-flg-field-pos-width flg reg-info)
    (let ((mask (1- (expt 2 size))))
      `(logand ,reg
               ,(ash mask pos)))))

; The N64 truncation below needs to be parameterized.

; Avoid loops in (at least) x86-update-reg-slice and x86-update-reg-field.
(local (in-theory (disable ash-to-floor ash-rewrite)))

(defun x86-update-reg-slice (flg val reg reg-info)
  (declare (xargs :guard (symbolp flg)))
  (mv-let (pos size)
    (x86-flg-field-pos-width flg reg-info)
    (let ((mask (1- (expt 2 size))))
      `(n64 ;; Extra mask -- prove away?  Hard because of (ASH ? POS).
        (logior
         ;; The bit(s) to insert
         (ash (logand ,mask ,val) ,pos)
         ;; Register with bit(s) cleared
         (logand ,reg
                 ,(lognot (ash mask pos))))))))
```

```
(defun x86-update-reg-field (flg val reg reg-info)
  (declare (xargs :guard (symbolp flg)))
  (mv-let (pos size)
    (x86-flg-field-pos-width flg reg-info)
    (let ((mask (1- (expt 2 size))))
      `(n64 ;; Extra mask -- prove away?  Hard because of (ASH ? POS).
        (logior
         ;; The bit(s) to insert -- already properly aligned
         ,val
         ;; Register with bit(s) cleared
         (logand ,reg
                 ,(lognot (ash mask pos))))))))

; Event to observe all events introduced by DEFSTOBJ.

(defun get-stobj-raw-defs (form state)
  (declare (xargs :mode :program :stobjs (state)))
  (let* ((name (cadr form))
         (args (cddr form))
         (wrld (w state))
         (template (defstobj-template name args wrld)))
    (defstobj-raw-defs name template wrld)))

; Nat-listp

(defun nat-listp (x)
  (declare (xargs :guard t))
  (if (atom x)
      (eq x nil)
    (and (natp (car x))
         (nat-listp (cdr x)))))

(defthm nat-listp-forward
  (implies (nat-listp x)
           (integer-listp x))
  :rule-classes :forward-chaining)

(defthm natp-nth
  (implies (and (nat-listp x)
                (integerp i)
                (<= 0 i)
                (< i (len x)))
           (and (integerp (nth i x))
                (<= 0 (nth i x))))
  :rule-classes :type-prescription)

(defthm nat-listp-forward-to-integer-listp
  (implies (nat-listp x)
           (integer-listp x))
  :rule-classes :forward-chaining)
```

```
; Convenient forcing idiom:

(defun formal-force-list (x)
  (declare (xargs :guard (true-listp x)))
  (cond ((endp x) nil)
        (t (cons `(force ,(car x))
                 (formal-force-list (cdr x))))))

(defmacro forced-and (&rest x)
  `(and ,@(formal-force-list x)))

; Handy utility for turning a positional list into an array

(defun list-to-alist (x i acc)
  (declare (xargs :guard (and (true-listp x)
                              (natp i)
                              (alistp acc))))
  (cond ((endp x) (reverse acc))
        (t (list-to-alist (cdr x)
                          (1+ i)
                          (acons i (car x) acc)))))

(defthm alistp-revappend
  (implies (true-listp x)
           (equal (alistp (revappend x y))
                  (and (alistp x)
                       (alistp y))))
  :hints (("Goal" :induct (revappend x y))))

(defthm alist-list-to-alist
  (implies (alistp acc)
           (alistp (list-to-alist x i acc))))

(defthm bounded-integer-alistp-monotone
  (implies (and (bounded-integer-alistp x i)
                (natp i)
                (natp j)
                (<= i j))
           (bounded-integer-alistp x j)))

(defthm bounded-integer-alistp-revappend
  (implies (true-listp x)
           (equal (bounded-integer-alistp (revappend x y) i)
                  (and (bounded-integer-alistp x i)
                       (bounded-integer-alistp y i))))
  :hints (("Goal" :induct (revappend x y))))

(defthm bounded-integer-alistp-list-to-alist
  (implies (and (natp i)
                (bounded-integer-alistp acc i)
                (equal k (+ i (len x))))
           (bounded-integer-alistp (list-to-alist x i acc)
                                   k)))

(defun list-to-array (name x)
  (declare (xargs :guard (and (symbolp name)
                              (true-listp x)
                              x
                              (< (length x)
                                 *maximum-positive-32-bit-integer*))))
  (let ((alist (list-to-alist x 0 nil))
        (len (length x)))
    (compress1 name
               `((:header :dimensions (,len)
                          :maximum-length ,(1+ len)
                          :default x
                          :name ,name)
                 ,@alist))))

(defun ints-to-booleans-acc (x acc)
; See ints-to-booleans.
  (declare (xargs :guard (and (integer-listp x)
                              (true-listp acc))))
  (cond ((endp x) (reverse acc))
        (t (ints-to-booleans-acc (cdr x)
                                 (cons (not (zerop (car x)))
                                       acc)))))

(defun ints-to-booleans (x)

; Maps a list of integers to a corresponding list of Booleans, treating 0 as
; false.  Example: (ints-to-booleans '(0 1 0 0 1)) ==>  (nil t  nil nil t).

  (declare (xargs :guard (integer-listp x)))
  (ints-to-booleans-acc x nil))

; More lemmas

(defthm logior-logand-inequality
; Initially for good-mem-tablep1-logic-bound-property
  (implies (forced-and (natp x)
                       (natp y)
                       (integerp z))
           (and (<= (logior x (logand y z))
                    (logior x y))
                (<= (logior x (logand z y))
                    (logior x y))))
  :hints (("Goal" :in-theory (enable logior-logand)))
  :rule-classes :linear)

(defthm logior-logand-inequality-commuted
  (implies (forced-and (natp x)
                      (natp y)
                      (integerp z))
           (and (<= (logior (logand y z) x)
                    (logior x y))
                (<= (logior (logand z y) x)
                    (logior x y))))
  :rule-classes :linear)

; redundant (in rtl/rel8/lib/log.lisp)
(defthm logand-commutative
  (equal (logand j i) (logand i j)))

; redundant (in rtl/rel8/lib/log.lisp)
(defthm logior-commutative
  (equal (logior j i) (logior i j)))

(defthm logand-with-2^48-1-is-identity
; Needed in guard proof for rm16
  (implies (and (natp x)
                (< x #x1000000000000)) ; 2^48
           (equal (logand #xffffffffffff x)
                  x))
  :hints (("Goal" :use ((:instance logand-expt-3 (x x) (n 48) (k 0)))
           :in-theory (enable bits))))

(defthm true-listp-revappend
  (equal (true-listp (revappend x y))
         (true-listp y)))

(encapsulate
 ()

 (local (include-book "arithmetic-5/top" :dir :system))
 (local (in-theory (enable ash-to-floor ash-rewrite)))

 (defthm ash-constant
   (implies (and (syntaxp (quote k))
                 (natp k)
                 (force (natp n)))
            (equal (ash n k)
                   (* (expt 2 k) n)))))

(defthm expt-positive
  (implies (and (rationalp r)
                (< 0 r)
                (integerp i))
           (and (rationalp (expt r i))
                (< 0 (expt r i))))
  :rule-classes :type-prescription)

(defthm integerp-expt
  (implies (and (natp r)
                (natp i))
           (natp (expt r i)))
  :rule-classes :type-prescription)
```

10 ========== File x86-64/operations.lisp ==========

```
; operations.lisp                        Warren A. Hunt, Jr.

(in-package "ACL2")

(include-book "misc-events")

(defmacro mk-name (&rest x)
  `(intern (concatenate 'string ,@x) "ACL2"))

(defun np-def-n (n)
  (declare (xargs :mode :program    ;; PACKN is a :program mode function
                  :guard (posp n)))
  (let* ((str-n          (symbol-name (if (< n 10)
                                          (packn (list 0 n))
                                        (packn (list n)))))
         (nXY            (mk-name "N" str-n))
         (nXY+           (mk-name "N" str-n "+"))
         (nXY-           (mk-name "N" str-n "-"))
         (nXYp           (mk-name "N" str-n "P"))
         (iXYp           (mk-name "I" str-n "P"))
         (ntoi           (mk-name "N" str-n "-TO-I" str-n))
         (ntoi-linear    (mk-name (symbol-name ntoi) "-LINEAR"))
         (ntoi-type      (mk-name (symbol-name ntoi) "-TYPE"))
         (iton           (mk-name "I" str-n "-TO-N" str-n))

         (nXYp-logxor-nXYp-less-than
          (mk-name "N" str-n "P-LOGXOR-N" str-n "P-LESS-THAN-2^" str-n))
         (nXYp-logior-nXYp-less-than
          (mk-name "N" str-n "P-LOGIOR-N" str-n "P-LESS-THAN-2^" str-n))
         )
    (list
     `(defmacro ,nXYp (x)
; Natural-number recognizer
        (list 'and
              (list 'integerp x)
              (list '<= 0 x)
              (list '< x ,(expt 2 n))))

     `(defmacro ,nXY (x)
; Natural-number truncation
        (list 'logand ,(1- (expt 2 n)) x))

     `(defmacro ,nXY+ (x y)
; Fixed-width, natural-number addition
        (list ',nXY (list '+ x y)))

     `(defmacro ,nXY- (x y)
; Fixed-width, natural-number subtraction
        (list ',nXY (list '- x y)))
```

```
`(defun ,iXYp (x)
   (declare (xargs :guard t))
   (and (integerp x)
        (<= ,(- (expt 2 (1- n))) x)
        (< x ,(expt 2 (1- n)))))

`(defun ,ntoi (x)
   (declare (xargs :guard (,nXYp x)))
   (if (< x ,(expt 2 (1- n))) x (- x ,(expt 2 n))))

`(defthm ,ntoi-linear
  (implies (force (,nXYp x))
           (and (<= ,(- (expt 2 (1- n))) (,ntoi x))
                (<= (,ntoi x) ,(1- (expt 2 (1- n))))))
  :rule-classes :linear)

`(defthm ,ntoi-type
   (implies (force (,nXYp x))
            (integerp (,ntoi x)))
   :rule-classes :type-prescription)

`(in-theory (disable ,ntoi))

`(defun ,iton (x)
   (declare (xargs :guard (,iXYp x)))
   (if (< x 0) (+ x ,(expt 2 n)) x))

`(in-theory (disable ,iXYp))

`(defthmd ,nXYp-logxor-nXYp-less-than
   (implies (and (,nXYp x)
                 (,nXYp y))
            (<= (logxor x y) ,(1- (expt 2 n))))
   :rule-classes :linear
   :hints
   (("Goal"
     :in-theory (disable logxor logxor-<=-expt-2-to-n)
     :use ((:instance logxor-<=-expt-2-to-n (n ,n))))))

`(defthmd ,nXYp-logior-nXYp-less-than
   (implies (and (,nXYp x)
                 (,nXYp y))
            (<= (logior x y) ,(1- (expt 2 n))))
   :rule-classes :linear
   :hints
   (("Goal"
     :in-theory (disable logior logior-less-than-2^n)
     :use ((:instance logior-less-than-2^n (n ,n))))))

`(table nxyp-expensive-linears
             t
             (append '(,nXYp-logior-nXYp-less-than
                       ,nXYp-logxor-nXYp-less-than)
                     (cdr (assoc t (table-alist 'nxyp-expensive-linears
                                                world)))))
     )))

(defmacro defuns-n ()
  (cons 'progn (np-def-n 1)))

; :trans (defuns-n)  ; For testing the NP-DEF-N macro

(defun np-defs (lst)
  (declare (xargs :mode :program
                  :guard (pos-listp lst)))
  (if (atom lst) nil
    (append (np-def-n (car lst))
            (np-defs (cdr lst)))))

(defmacro defuns-np (&rest lst)
  (cons 'progn (np-defs lst)))

(defuns-np 1 2 3 4 5 8 12 16 18 20 24 27 30 32 45 48 64 120 128)

(deftheory nxyp-expensive-linears
  (cdr (assoc t (table-alist 'nxyp-expensive-linears world))))

; It is expected that all lemmas directly dealing with the functions
; have been proven -- so these function are disabled.

(in-theory (disable logand))
(in-theory (disable logxor))
(in-theory (disable logior))

(with-arithmetic-help-5
 (defthm ash-n02p-is-zero-or-positive
   (implies (natp x)
            (<= 0 (ash x n)))
   :rule-classes :linear))

(in-theory (disable ash))

; Finally, here are some related definitions that we find useful.

(defun n04p? (x)
  (declare (xargs :guard t))
  (or (n04p x)
      (null x)))

(defun n08p? (x)
  (declare (xargs :guard t))
  (or (n08p x)
      (null x)))

(defun n64p? (x)
  (declare (xargs :guard t))
  (or (n64p x)
      (null x)))

(defun bytesp (nbytes x)
  (declare (xargs :guard (natp nbytes)))
  (and (natp x)
       (let ((nbits (ash nbytes 3)))
         (< x (ash 1 nbits)))))

(defthm bytesp-forward
  (implies (bytesp n x)
           (natp x))
  :rule-classes :forward-chaining)

(defthm bytesp-0
  (equal (bytesp 0 x)
         (equal x 0)))

(defthm bytesp-x-0
  (implies (force (natp x))
           (bytesp x 0))
  :hints (("Goal" :in-theory (enable bytesp))))

(defthm bytesp-1-forward
  (implies (bytesp 1 x)
           (<= x #xff))
  :rule-classes :forward-chaining)

(defthm bytesp-2-forward
  (implies (bytesp 2 x)
           (<= x #xffff))
  :rule-classes :forward-chaining)

(defthm bytesp-4-forward
  (implies (bytesp 4 x)
           (<= x #xffffffff))
  :rule-classes :forward-chaining)

(defthm bytesp-8-forward
  (implies (bytesp 8 x)
           (<= x #uxffffffff_ffffffff))
  :rule-classes :forward-chaining)

(defthm bytesp-backward
  (implies (and (syntaxp (quotep nbytes))
                (natp nbytes)
                (natp x)
                (< x (ash 1 (ash nbytes 3))))
           (bytesp nbytes x)))

(defthm bytesp-logand-fs
  (implies (force (natp nbytes))
           (bytesp nbytes
                   (logand (+ -1 (expt 2 (* 8 nbytes)))
                           k))))

(defthm bytesp-logand-fs-commuted
  (implies (force (natp nbytes))
           (bytesp nbytes
                   (logand k
                           (+ -1 (expt 2 (* 8 nbytes)))))))

(in-theory (disable bytesp))

(defun n+ (bits x y)
  (declare (xargs :guard (and (posp bits)
                              (integerp x)
                              (integerp y))))
  (logand (+ x y)
          (1- (ash 1 bits))))

(defun nx (bits x)
  (declare (xargs :guard (and (natp bits)
                              (integerp x))))
  (logand x (1- (ash 1 bits))))

(defmacro natp? (x)
  `(or (equal ,x nil)
       (natp ,x)))
```

11 ========== File x86-64/x86-state.lisp ==========

```
;;; !!! TO DO:

;;; Maybe remove need for hack to get around guard-checking penalty:
#||
(redef)
(defun x86-64p (x86-64)
  (declare (xargs :stobjs x86-64))
  (x86-64p-pre x86-64))
||#

;;; Complete updating the x86-64 stobj for 64 bits.  In particular, extend some
;;; registers to 64-bit and add some registers.  Consider restoring these:
;;; seg-base, seg-limit, and seg-access.

;;; (Low priority) Consider disabling linear rules.  Some profiling done on a
;;; version of x86-64p-!rgfi-n03p, replacing x86-64p by x86-64p-pre, showed
;;; considerable time in ADD-TERMS-AND-LEMMAS etc.

;;; Look for include-book forms and find lemmas that should be moved to
;;; misc-events.lisp.

;;; From Matt to self: Consider fixing make-event (actually checking of
;;; embedded-event-form for record-expansoin calls) so that I can make the
;;; include-book of proof-by-arith be local (see note below about
;;; record-expansion).

;;; Enhance the test, test-x86-64, perhaps to force resizing and ultimately to
;;; use up all pages.

;;; Perhaps speed up initialize-x86-64 (which takes Matt about a second) by
;;; saving the list of addresses that have been allocated.  Or, avoid
;;; initializing all of the mem-array -- just initialize all pages with entries
;;; from mem-table (initializing all 2^18 quadword values in all such pages).
;;; ACTUALLY, just think -- probably it suffices to zero out the array up to
;;; *initial-mem-array-length* -- consider replacing
;;; (initialize-mem-array 0 x86-64) by (initialize-mem-array
;;; *initial-mem-array-length* x86-64) in initialize-x86-64.

;;; END of "TO DO"

; x86-state.lisp                              Warren A. Hunt, Jr.

; We model the X86 state with several arrays: two arrays for the memory and
; additional arrays for various register files.  This data structure holds all
; of the supervisor data as well.

(in-package "ACL2")

(include-book "constants")
```

```
; We could increase memory for X86 memory, as shown below.  However,
; preliminary experiments suggest that this does not speed things up.
; (include-book "centaur/misc/memory-mgmt-logic" :dir :system)
; (value-triple (set-max-mem (* 6 (expt 2 30))))

; Here we can include the GL book to help verify various arithmetic facts,
; if necessary.
; (local (include-book "centaur/gl/gl" :dir :system))

(defun gl-int (start by count)
  (declare (xargs :guard (and (natp start)
                              (natp by)
                              (natp count))))
  (if (zp count)
      nil
    (cons start
          (gl-int (+ by start) by (1- count)))))


(defstobj x86-64

; Originally the array fields used UNSIGNED-BYTE types rather than types such
; as (SATISFIES N32P) and (SATISFIES N08P), so that we could help the host Lisp
; system allocate these arrays using minimal real (physical) memory.  However,
; some performance experiments suggest avoiding type declarations altogether.

; NOTE: We've reverted to using type declarations in the stobjs array fields,
; even though CCL time performance might not be as good this way.  The reason
; is a bit subtle.  Basically, those types go into the definition of functions
; like rgfp, which go into the definition of x86-64p.  We can't easily define
; our own versions unless we make them non-executable, because we can't get
; those array "handles" except via non-executable code.  But we use x86-64p in
; our guards, and we want it to be executable.  If performance becomes an
; issue, we should simply modify ACL2 by adding a new keyword for stobjs array
; fields, say :lisp-type, which can be weaker than the :type field and is used
; in the raw Lisp code.

  ;; The model register file has a simple structure; it just an array
  ;; of 64-bit words.
  (rgf :type (array (unsigned-byte 64)
                    (*m86-64-reg-names-len*))
       :initially 0
       :resizable nil)

  ;; the program counter
  (rip :type (unsigned-byte 64)
       :initially 0)

  ;; the eflags register
  (flg :type (unsigned-byte 64)
       :initially 0)

  ;; the segment registers
  (seg :type (array (unsigned-byte 16)
                    (*m86-64-segment-reg-names-len*))
       :initially 0
       :resizable nil)

  ;; The System Table Registers (GDTR and IDTR) point to bounded
  ;; tables of (up to 8192) segment descriptors.
  (str :type (array (unsigned-byte 48)
                    (*m86-64-gdtr-idtr-names-len*))
       :initially 0
       :resizable nil)

  ;; These 16-bit values are Segment Selectors (Task Register and
  ;; LDTR):  Index(12),TI(1),RPL(2).  The Index references a segment
  ;; descriptor in the Global Descriptor Table (GDT).
  (ssr :type (array (unsigned-byte 16)
                    (*m86-64-ldtr-tr-names-len*))
       :initially 0
       :resizable nil)

  ;; the control registers
  (ctr  :type (array (unsigned-byte 64)
                     (*m86-64-control-reg-names-len*))
       :initially 0
       :resizable nil)

  ;; the debug registers
  (dbg :type (array (unsigned-byte 64)
                    (*m86-64-debug-reg-names-len*))
       :initially 0
       :resizable nil)

  ;; Additional registers are to be defined here.

  ;; FPU 16-bit status registers

  ;; FPU 48-bit last instruction registers

  ;; FPU 80-bit data registers

  ;; XMM 16-bit status

  ;; XMM 128-bit data registers

  ;; Our model is intended to represent a 2^48 byte virtual memory.  We choose
  ;; (for performance reasons) to model the 2^48 byte memory as a 2^45 quadword
  ;; (8-byte) memory, with one level of indirection.  We maintain the
  ;; byte-level semantics.
```

```
; The following comment also appears in constants.lisp; keep these in sync.

; The memory is laid out in a flat array, to be viewed as back-to-back 2MB
; pages.  Here is an example of how that works.  Suppose we have:

;   mem-table[#x7654321] = 0_[18-zeros]
;   mem-table[#x36789ab] = 1_[18-zeros]
;   mem-table[#x2111111] = 2_[18-zeros]

; Then mem-array starts as follows.  Of course, 2^18 qwords = 2^21 bytes, i.e.,
; 2 MB.

;   2^18 qwords corresponding to qword addresses with top 27 bits = #x7654321
;   2^18 qwords corresponding to qword addresses with top 27 bits = #x36789ab
;   2^18 qwords corresponding to qword addresses with top 27 bits = #x2111111

; All additional values in mem-table are the initial value of 1, which means
; "page is not present".

  (mem-table :type
             (array (unsigned-byte 45) ; 1, or qword addr w/ low 18 bits of 0
                    (*mem-table-size*))
             :initially 1
             :resizable nil)

  (mem-array :type (array (unsigned-byte 64)
                          (*initial-mem-array-length*))
             :initially 0 ; *default-mem-value*
             :resizable t)

  (mem-array-next-addr :type

; We briefly considered type t here, in order to avoid a guard burden when we
; bump this field upward in add-page.  However, mem-array-next-addr ultimately
; goes into the mem-table, so we'll need to know that it's an (unsigned-byte
; 45) -- so we might as well go ahead and say so here, which avoids having to
; say so in good-memp.

                       (integer 0 35184372088832) ; (expt 2 45)
                       :initially 0)

#|| Relics of 32-bit model -- might need to be restored suitably.

  ;; This information is loaded from memory when a segment register is loaded.
  ;; For each segment register, there is a corresponding "seg-base", "seg-limit",
  ;; and "seg-access" register; this memory-resident information is automatically
  ;; used when protection is enabled.
  (seg-base   :type (array (unsigned-byte 32)
                          (*m86-64-segment-reg-names-len*))
              :initially 0 :resizable nil)
  (seg-limit  :type (array (unsigned-byte 20)
                           (*m86-64-segment-reg-names-len*))
              :initially 0 :resizable nil)
  (seg-access :type (array (unsigned-byte 12)
                           (*m86-64-segment-reg-names-len*))
              :initially 0 :resizable nil)
||#

  ;; Here we have some additional state -- normally stored in the
  ;; X86 memory, but we make a local copy for performance reasons.
  ;; The X86 processor does something similar by reading such data
  ;; and then loading it into internal X86 processor registers.

  ;; The state of the ACL2 model.  This flag is not part of the X86
  ;; processor; it is used to signal problems with model state, such
  ;; as the processor is halted.  While this flag is NIL, the processor
  ;; model is OK; otherwise, the flag indicates (part of) the problem.
  (ms :type t :initially nil)
  :inline t
  :renaming
  ((x86-64p x86-64p-pre)
   (update-rgfi !rgfi)
   (update-rip !rip)
   (update-flg !flg)
   (update-segi !segi)
   (update-stri !stri)
   (update-ssri !ssri)
   (update-ctri !ctri)
   (update-dbgi !dbgi)
   (update-mem-tablei !mem-tablei)
   (update-mem-arrayi !mem-arrayi)
   (update-mem-array-next-addr !mem-array-next-addr)
#|| Relics of 32-bit model -- might need to be restored suitably.
   (update-seg-basei !seg-basei)
   (update-seg-limiti !seg-limiti)
   (update-seg-accessi !seg-accessi)
||#
   (update-ms !ms)

; Additions in case we define our own array field recognizers:
#||
   (rgfp rgfp-pre)
   (segp segp-pre)
   (strp strp-pre)
   (ssrp ssrp-pre)
   (ctrp ctrp-pre)
   (dbgp dbgp-pre)
   (mem-tablep mem-tablep-pre)
   (mem-arrayp mem-arrayp-pre)
   (seg-basep seg-basep-pre)
   (seg-limitp seg-limitp-pre)
   (seg-accessp seg-accessp-pre)
||#

   ))

; MEM-TABLE read lemmas

(defthm mem-tablep-forward
  (implies (mem-tablep x)
           (nat-listp x))
  :rule-classes :forward-chaining)

(defmacro mem-table-indexp (x)
  `(and (natp ,x)
        (< ,x *mem-table-size*)))

(defthm natp-mem-tablep-when-valid-mem-table-index
  (implies (forced-and (x86-64p-pre x86-64)
                       (mem-table-indexp i))
           (and (integerp (mem-tablei i x86-64))
                (<= 0 (mem-tablei i x86-64))))
  :rule-classes (:rewrite :type-prescription))

(defthm natp-mem-tablep-when-valid-mem-table-index-nth-version
  (implies (forced-and (x86-64p-pre x86-64)
                       (mem-table-indexp i))
           (and (integerp (nth i (nth *mem-tablei* x86-64)))
                (<= 0 (nth i (nth *mem-tablei* x86-64)))))
  :rule-classes (:rewrite :type-prescription))

(encapsulate
 ()

 (local (defthm nth-of-mem-table-<=-expt-2-45
          (implies (and (mem-tablep x)
                        (integerp i)
                        (<= 0 i)
                        (< i (len x)))
                   (< (nth i x) *2^45*))
          :rule-classes :linear
          :hints (("Goal" :in-theory (e/d (nth) ())))))

 (defthm mem-tablei-less-than-expt-2-45
   (implies (forced-and (x86-64p-pre x86-64)
                        (mem-table-indexp i))
            (< (mem-tablei i x86-64) *2^45*))
   :rule-classes :linear)

 (defthm mem-tablei-less-than-expt-2-45-nth-version
   (implies (forced-and (x86-64p-pre x86-64)
                        (mem-table-indexp i))
            (< (nth i (nth *mem-tablei* x86-64))
               *2^45*))
   :rule-classes :linear))

; Replace logior by +

; Idea for proving logior-logand-2^18 (which comes later, but we need two of
; its lemmas now):

; Let x' = x/2^18 and z' = z/2^18.  Since x < z, x' < z'.
; (logior x (logand y #x3ffff)) <= (logior x #x3ffff) {LOGIOR-LOGAND-INEQUALITY}
; = (logior (x' * 2^18) #x3ffff) = (+ (x' * 2^18) #x3ffff) {LOGIOR-EXPT}
; < (x' + 1) * 2^18
; <= z' * 2^18 = z

(encapsulate
 ()

 (local (include-book "arithmetic-5/top" :dir :system))

 (defthm logior-logand-2^18-1
   (implies (and (natp x)
                 (equal (logand #x3ffff x) 0))
            (equal x (* *2^18* (floor x *2^18*))))
   :rule-classes nil))

(encapsulate
 ()

 (local (include-book "rtl/rel8/lib/top" :dir :system))

 (defthm logior-logand-2^18-2
   (implies (and (natp x1)
                 (n18p y))
            (equal (logior (* *2^18* x1) y)
                   (+ (* *2^18* x1) y)))
   :hints (("Goal"
            :in-theory (enable bvecp)
            :use ((:instance logior-expt
                             (n 18) (x x1)))))))

(defthm logior-with-multiple-of-2^18
  (implies (and (natp x)
                (equal (logand #x3ffff x) 0)
                (n18p y))
           (equal (logior x y)
                  (+ x y)))
  :hints (("Goal"
           :use (logior-logand-2^18-1
                 (:instance logior-logand-2^18-2
                            (x1 (floor x *2^18*)))))))
```

```
(defthm logior-with-multiple-of-2^18-commuted
  (implies (and (natp x)
                (equal (logand #x3ffff x) 0)
                (n18p y))
           (equal (logior y x)
                  (+ x y))))

; x86-64p

(defun good-mem-table-entriesp-exec (i table-max-index array-next-addr x86-64)
  (declare (type (unsigned-byte 27) ; *mem-table-size*
                 i table-max-index)
           (xargs :stobjs x86-64
                  :guard (and (natp array-next-addr)
                              (<= i table-max-index))
                  :measure (nfix (- table-max-index i))))
  (cond ((mbt (and (natp i) (natp table-max-index) (<= i table-max-index)))
         (let ((addr (mem-tablei i x86-64)))
           (and (or (eql addr 1)
                    (and (natp addr)
                         (eql (logand #x3ffff addr) 0)
                         (< (+ #x3ffff addr)

; We could use addr instead of (+ #x3ffff addr) just above, since presumably
; array-next-addr is always a multiple of 2^18.  Of course, some proofs would
; likely change.

                            array-next-addr)))
                (or (eql i table-max-index)
                    (good-mem-table-entriesp-exec
                     (1+ i) table-max-index array-next-addr x86-64)))))
        (t nil)))

(defun good-mem-table-entriesp-logic (i table-max-index array-next-addr mem-table)
  (declare (xargs :measure (nfix (- table-max-index i))))
  (cond ((mbt (and (natp i) (natp table-max-index) (<= i table-max-index)))
         (let ((addr (nth i mem-table)))
           (and (or (eql addr 1)
                    (and (natp addr)
                         (eql (logand #x3ffff addr) 0)
                         (< (+ #x3ffff addr) array-next-addr)))
                (or (eql i table-max-index)
                    (good-mem-table-entriesp-logic
                     (1+ i) table-max-index array-next-addr mem-table)))))
        (t nil)))

(defun-nx nth-nx (n x)
  (nth n x))

(defthm good-mem-table-entriesp-exec-is-good-mem-table-entriesp-logic
  (equal (good-mem-table-entriesp-exec i table-max-index array-next-addr x86-64)
         (good-mem-table-entriesp-logic i table-max-index array-next-addr
                                        (nth *mem-tablei* x86-64))))

(in-theory (disable good-mem-table-entriesp-logic))

(defun merge-<-into-> (lst1 lst2 acc)

; Merge upward-sorted lists lst1 and lst2 into downward-sorted list acc, to
; produce a downward-sorted list.

  (declare (xargs :guard (and (rational-listp lst1)
                              (rational-listp lst2))
                  :measure (+ (len lst1) (len lst2))))
  (cond ((endp lst1) (revappend lst2 acc))
        ((endp lst2) (revappend lst1 acc))
        ((< (car lst1) (car lst2))
         (merge-<-into-> (cdr lst1) lst2 (cons (car lst1) acc)))
        (t
         (merge-<-into-> lst1 (cdr lst2) (cons (car lst2) acc)))))

(defun merge->-into-< (lst1 lst2 acc)

; Merge downward-sorted lists lst1 and lst2 into upward-sorted list acc, to
; produce an upward-sorted list.

  (declare (xargs :guard (and (rational-listp lst1)
                              (rational-listp lst2))
                  :measure (+ (len lst1) (len lst2))))
  (cond ((endp lst1) (revappend lst2 acc))
        ((endp lst2) (revappend lst1 acc))
        ((> (car lst1) (car lst2))
         (merge->-into-< (cdr lst1) lst2 (cons (car lst1) acc)))
        (t
         (merge->-into-< lst1 (cdr lst2) (cons (car lst2) acc)))))

(defun good-mem-table-entriesp-weak (i table-max-index x86-64)

; For guard of mem-table-entries.

  (declare (type (unsigned-byte 27) ; *mem-table-size*
                 i table-max-index)
           (xargs :stobjs x86-64
                  :guard (<= i table-max-index)
                  :measure (nfix (- table-max-index i))))
  (cond ((mbt (and (natp i) (natp table-max-index) (<= i table-max-index)))
         (and (natp (mem-tablei i x86-64))
              (or (eql i table-max-index)
                  (good-mem-table-entriesp-weak (1+ i) table-max-index x86-64))))
        (t nil)))
```

```
(encapsulate
 ()

 (local (include-book "arithmetic-5/top" :dir :system))

 (defun mem-table-entries (lower upper x86-64 parity)
   (declare (type (unsigned-byte 27) ; *mem-table-size*
                  lower upper)
            (xargs :stobjs x86-64
                   :guard
                   (and (<= lower upper)
                        (booleanp parity)
                        (good-mem-table-entriesp-weak lower upper x86-64))
                   :verify-guards nil
                   :measure (nfix (- upper lower))))
   (cond ((eql lower upper)
          (let ((addr (mem-tablei lower x86-64)))
            (cond ((eql addr 1) nil)
                  (t (list addr)))))
         ((eql (1+ lower) upper)
          (let ((addr-lower (mem-tablei lower x86-64))
                (addr-upper (mem-tablei upper x86-64)))
            (cond ((eql addr-lower 1)
                   (cond ((eql addr-upper 1) nil)
                         (t (list addr-upper))))
                  ((eql addr-upper 1)
                   (list addr-lower))
                  ((equal parity (< addr-lower addr-upper))
                   (list addr-lower addr-upper))
                  (t
                   (list addr-upper addr-lower)))))
         ((mbt (and (natp lower) (natp upper) (< (1+ lower) upper)))
          (let ((mid (ash (+ lower upper) -1)))
            (cond (parity
                   (merge->-into-<
                    (mem-table-entries lower mid x86-64 nil)
                    (mem-table-entries (1+ mid) upper x86-64 nil)
                    nil))
                  (t
                   (merge-<-into->
                    (mem-table-entries lower mid x86-64 t)
                    (mem-table-entries (1+ mid) upper x86-64 t)
                    nil)))))
         (t 'impossible))))

(defthm rational-listp-revappend
  (implies (rational-listp x)
           (equal (rational-listp (revappend x y))
                  (rational-listp y))))

(defthm rational-listp-merge->-into-<
  (implies (and (rational-listp x)
                (rational-listp y)
                (rational-listp z))
           (rational-listp (merge->-into-< x y z))))

(defthm rational-listp-merge-<-into->
  (implies (and (rational-listp x)
                (rational-listp y)
                (rational-listp z))
           (rational-listp (merge-<-into-> x y z))))

(encapsulate
 ()

 (local (defthm good-mem-table-entriesp-weak-preserved-lemma
          (implies (and (good-mem-table-entriesp-weak lower upper1 x86-64)
                        (natp upper2)
                        (<= lower upper2)
                        (<= upper2 upper1))
                   (good-mem-table-entriesp-weak lower upper2 x86-64))
          :hints (("Goal" :in-theory (disable (force))))))

 (defthm good-mem-table-entriesp-weak-preserved
   (implies (and (good-mem-table-entriesp-weak lower1 upper1 x86-64)
                 (natp lower2)
                 (natp upper2)
                 (<= lower1 lower2)
                 (<= lower2 upper2)
                 (<= upper2 upper1))
            (good-mem-table-entriesp-weak lower2 upper2 x86-64))
   :hints (("Goal" :in-theory (disable (force))))))

(encapsulate
 ()
 (local (include-book "arithmetic-5/top" :dir :system))

 (defthm ash-minus-1-inequality-1
   (implies (and (natp lower)
                 (natp upper)
                 (< (1+ lower) upper))
            (< lower
               (ash (+ lower upper) -1)))
   :rule-classes :linear)

 (defthm ash-minus-1-inequality-2
   (implies (and (natp lower)
                 (natp upper)
                 (< (1+ lower) upper))
            (<= (+ 1 (ash (+ lower upper) -1))
                upper))
   :rule-classes :linear))
```

```
(defthm rational-listp-mem-table-entries
  (implies (good-mem-table-entriesp-weak lower upper x86-64)
           (rational-listp (mem-table-entries lower upper x86-64 parity)))
  :hints (("Goal"
           :induct (mem-table-entries lower upper x86-64 parity)
           :in-theory (disable (force)))))

(verify-guards mem-table-entries)

(defun no-duplicatesp-sorted (lst)
  (declare (xargs :guard (eqlable-listp lst)))
  (cond ((or (endp lst)
             (endp (cdr lst)))
         t)
        ((eql (car lst) (cadr lst))
         nil)
        (t (no-duplicatesp-sorted (cdr lst)))))

(local (defthm rational-listp-implies-eqlable-listp
         (implies (rational-listp x)
                  (eqlable-listp x))))

(defun good-mem-table-no-dupsp-exec (lower upper x86-64)
  (declare (type (unsigned-byte 27) ; *mem-table-size*
                 lower upper)
           (xargs :stobjs x86-64
                  :guard
                  (and (<= lower upper)
                       (good-mem-table-entriesp-weak lower upper x86-64))))
  (no-duplicatesp-sorted (mem-table-entries lower upper x86-64 t)))

(encapsulate
 ()

 (local (include-book "arithmetic-5/top" :dir :system))

 (defun mem-table-entries-logic (lower upper mem-table parity)
; The result is increasing if parity is true, increasing if parity is false.
   (declare (xargs :measure (nfix (- upper lower))))
   (cond ((eql lower upper)
          (let ((addr (nth lower mem-table)))
            (cond ((eql addr 1) nil)
                  (t (list addr)))))
         ((eql (1+ lower) upper)
          (let ((addr-lower (nth lower mem-table))
                (addr-upper (nth upper mem-table)))
            (cond ((eql addr-lower 1)
                   (cond ((eql addr-upper 1) nil)
                         (t (list addr-upper))))
                  ((eql addr-upper 1)
                (list addr-lower))
               ((equal parity (< addr-lower addr-upper))
                (list addr-lower addr-upper))
               (t
                (list addr-upper addr-lower)))))
         ((mbt (and (natp lower) (natp upper) (< (1+ lower) upper)))
          (let ((mid (ash (+ lower upper) -1)))
            (cond (parity
                   (merge->-into-<
                    (mem-table-entries-logic lower mid mem-table nil)
                    (mem-table-entries-logic (1+ mid) upper mem-table nil)
                    nil))
                  (t
                   (merge-<-into->
                    (mem-table-entries-logic lower mid mem-table t)
                    (mem-table-entries-logic (1+ mid) upper mem-table t)
                    nil)))))
         (t 'impossible))))

(defun good-mem-table-no-dupsp-logic (lower upper mem-table)
  (no-duplicatesp-sorted (mem-table-entries-logic lower upper mem-table t)))

(defthm mem-table-entries-is-mem-table-entries-logic
  (equal (mem-table-entries lower upper x86-64 parity)
         (mem-table-entries-logic lower upper
                                  (nth *mem-tablei* x86-64)
                                  parity)))

(defthm good-mem-table-no-dupsp-is-good-mem-table-no-dupsp-logic
  (equal (good-mem-table-no-dupsp-exec lower upper x86-64)
         (good-mem-table-no-dupsp-logic lower upper
                                        (nth *mem-tablei* x86-64))))

(in-theory (disable good-mem-table-no-dupsp-logic
                    good-mem-table-no-dupsp-exec))

; Before defining good-memp, towards defining x86-64p, we define a function
; that will let us reason about the mem-array-next-addr field, showing (e.g.,
; for guard verification) that it's not too large.

(defun expected-mem-array-next-addr (i table-len x86-64)
  (declare (type (integer 0 134217728) ; *2^27* = *mem-table-size*
                 i table-len)
           (xargs :stobjs x86-64
                  :guard (<= i table-len)
                  :measure (nfix (- table-len i))))
  (cond ((or (not (natp i))
             (not (natp table-len))
             (>= i table-len))
         0)
        (t (let ((addr (mem-tablei i x86-64)))
             (cond ((eql addr 1)
                    (expected-mem-array-next-addr (1+ i) table-len x86-64))
                   (t (+ *2^18*
                         (expected-mem-array-next-addr (1+ i)
                                                       table-len
                                                       x86-64))))))))

(defthm expected-mem-array-in-parts
  (implies (and (natp i)
                (natp j)
                (natp k)
                (<= i j)
                (<= j k))
           (equal (+ (expected-mem-array-next-addr i j x86-64)
                     (expected-mem-array-next-addr j k x86-64))
                  (expected-mem-array-next-addr i k x86-64)))
  :rule-classes nil)

; Note to myself from Matt: Could be local if chk-embedded-event-form doesn't
; look at first argument of record-expansion.
; Actually no longer needed, for now at least:
; (include-book "make-event/proof-by-arith" :dir :system)

(encapsulate
 ()

 (local (include-book "arithmetic-5/top" :dir :system))

 (defthm expected-mem-array-bound-general
   (implies (<= i table-len)
            (<= (expected-mem-array-next-addr i table-len x86-64)
                (* *2^18* (- table-len i))))

; Need linear rule here, rather than :rule-classes nil, for later lemma
; expected-mem-array-next-addr-bound-general.

   :rule-classes :linear))

(defthm expected-mem-array-bound
  (implies (<= i table-len)
           (<= (expected-mem-array-next-addr 0
                                             (mem-table-length x86-64)
                                             x86-64)
               (* *2^18* (mem-table-length x86-64))))
  :hints (("Goal" :use ((:instance expected-mem-array-bound-general
                                   (i 0)
                                   (table-len (mem-table-length x86-64))))))
  :rule-classes :linear)

(in-theory (disable expected-mem-array-next-addr))
```

```
(defthm good-mem-table-entriesp-logic-forward-to-good-mem-table-entriesp-weak
  (implies (good-mem-table-entriesp-logic lower upper
                                          array-next-addr
                                          (nth *mem-tablei* x86-64))
           (good-mem-table-entriesp-weak lower upper x86-64))
  :hints (("Goal" :in-theory (e/d (good-mem-table-entriesp-logic
                                   good-mem-table-entriesp-weak)
                                  ((force)))))
  :rule-classes :forward-chaining)

(defun good-mem-arrayp-1 (index len x86-64)
  (declare (xargs :stobjs x86-64
                  :guard (and (natp index)
                              (natp len)
                              (<= index len)
                              (<= len (mem-array-length x86-64)))
                  :measure (nfix (- len index))))
  (cond ((mbe :logic (not (and (natp index)
                               (natp len)
                               (< index len)))
              :exec (eql index len))
         t)
        (t (and (eql (mem-arrayi index x86-64) 0)
                (good-mem-arrayp-1 (1+ index) len x86-64)))))

(defun good-mem-arrayp-1-logic (index len mem-array)
  (declare (xargs :measure (nfix (- len index))))
  (cond ((not (and (natp index)
                   (natp len)
                   (< index len)))
         t)
        (t (and (eql (nth index mem-array) 0)
                (good-mem-arrayp-1-logic (1+ index) len mem-array)))))

(defthm good-mem-arrayp-1-is-good-mem-arrayp-1-logic
  (equal (good-mem-arrayp-1 index len x86-64)
         (good-mem-arrayp-1-logic index len (nth *mem-arrayi* x86-64))))

(defun good-mem-arrayp (x86-64)
  (declare (xargs :stobjs x86-64
                  :guard (<= (mem-array-next-addr x86-64)
                             (mem-array-length x86-64))))
  (mbe :logic
       (good-mem-arrayp-1-logic (mem-array-next-addr x86-64)
                                (mem-array-length x86-64)
                                (nth-nx *mem-arrayi* x86-64))
       :exec
       (good-mem-arrayp-1 (mem-array-next-addr x86-64)
                          (mem-array-length x86-64)
                          x86-64)))
```

```
(defmacro good-mem-table-entriesp (i table-max-index array-next-addr x86-64-var)
  `(mbe :logic
        (good-mem-table-entriesp-logic ,i ,table-max-index ,array-next-addr
                                       (nth-nx *mem-tablei* ,x86-64-var))
        :exec
        (good-mem-table-entriesp-exec ,i ,table-max-index ,array-next-addr
                                      ,x86-64-var)))

(defmacro good-mem-table-no-dupsp (i table-max-index x86-64-var)
  `(mbe :logic
       (good-mem-table-no-dupsp-logic ,i ,table-max-index
                                      (nth-nx *mem-tablei* ,x86-64-var))
       :exec
       (good-mem-table-no-dupsp-exec ,i ,table-max-index ,x86-64-var)))

(defun good-memp (x86-64)
  (declare (xargs :stobjs x86-64))
  (let ((table-max-index (1- (mem-table-length x86-64)))
        (array-length (mem-array-length x86-64))
        (array-next-addr (mem-array-next-addr x86-64)))
    (and (<= array-next-addr array-length)
         (<= *initial-mem-array-length* array-length)
         (eql (logand #x3ffff array-length) 0) ; integral number of pages
         (eql array-next-addr
              (expected-mem-array-next-addr 0
                                            (mem-table-length x86-64)
                                            x86-64))
         (good-mem-table-entriesp 0 table-max-index array-next-addr x86-64)
         (good-mem-table-no-dupsp 0 table-max-index x86-64)
         (good-mem-arrayp x86-64))))

(defun x86-64p (x86-64)
  (declare (xargs :stobjs x86-64))
  (and (x86-64p-pre x86-64)
       (good-memp x86-64)))

; Lemmas to help with proofs about STOBJs that hold X86 state.  Our
; goal is to prove a nice set of forward-chaining lemmas, as our
; predicates seem nicely set up for that.

(in-theory (disable nth))  ; Because NTH used to select object from
                           ; the x86-64 state.

; Lemmas below are in the same order as the fields declare in the
; X86-64 STOBJ above.

; We first deal with the STOBJ read lemmas

; RGF read lemmas

(defthm rgfp-forward
  (implies (rgfp x)
           (nat-listp x))
  :rule-classes :forward-chaining)

(defthm natp-rgfi
  (implies (forced-and (x86-64p x86-64)
                       (n04p i))
           (and (integerp (rgfi i x86-64))
                (<= 0 (rgfi i x86-64))))
  :rule-classes :type-prescription)

(encapsulate
 ()

 (local (defthm nth-of-rgf-<=expt-2-64
          (implies (and (rgfp x)
                        (integerp i)
                        (<= 0 i)
                        (< i (len x)))
                   (< (nth i x) *2^64*))
          :rule-classes :linear
          :hints (("Goal" :in-theory (e/d (nth) ())))))

 (defthm rgfi-less-than-expt-2-64
   (implies (forced-and (x86-64p x86-64)
                        (n04p i))
            (< (rgfi i x86-64) *2^64*))
   :rule-classes :linear))

; RIP read lemmas

(defthm natp-rip
  (implies (force (x86-64p x86-64))
           (and (integerp (rip x86-64))
                (<= 0 (rip x86-64))))
  :rule-classes :type-prescription)

(defthm rip-less-than-expt-2-64
  (implies (force (x86-64p x86-64))
           (< (rip x86-64) *2^64*))
  :rule-classes :linear)


; FLG read lemmas

(defthm natp-flg
  (implies (force (x86-64p x86-64))
           (and (integerp (flg x86-64))
                (<= 0 (flg x86-64))))
  :rule-classes :type-prescription)
```

```
(defthm flg-less-than-expt-2-64
  (implies (x86-64p x86-64)
           (< (flg x86-64) *2^64*))
  :rule-classes :linear)


; SEG read lemmas

(defthm segp-forward
  (implies (segp x)
           (nat-listp x))
  :rule-classes :forward-chaining)

(defthm natp-segi
  (implies (forced-and (x86-64p x86-64)
                       (n01p i))
           (and (integerp (segi i x86-64))
                (<= 0 (segi i x86-64))))
  :rule-classes :type-prescription)

(encapsulate
 ()

 (local (defthm nth-of-seg-<=expt-2-16
          (implies (and (segp x)
                        (integerp i)
                        (<= 0 i)
                        (< i (len x)))
                   (< (nth i x) *2^16*))
          :rule-classes :linear
          :hints (("Goal" :in-theory (e/d (nth) ())))))

 (defthm segi-less-than-expt-2-16
   (implies (forced-and (x86-64p x86-64)
                        (n01p i))
            (< (segi i x86-64) *2^16*))
   :rule-classes :linear))

; STR read lemmas

(defthm strp-forward
  (implies (strp x)
           (nat-listp x))
  :rule-classes :forward-chaining)

(defthm natp-stri
  (implies (forced-and (x86-64p x86-64)
                       (n01p i))
           (and (integerp (stri i x86-64))
                (<= 0 (stri i x86-64))))
  :rule-classes :type-prescription)
```

```
(encapsulate
 ()

 (local (defthm nth-of-str-<=expt-2-48
          (implies (and (strp x)
                        (integerp i)
                        (<= 0 i)
                        (< i (len x)))
                   (< (nth i x) *2^48*))
          :rule-classes :linear
          :hints (("Goal" :in-theory (e/d (nth) ())))))

 (defthm stri-less-than-expt-2-48
   (implies (forced-and (x86-64p x86-64)
                        (n01p i))
            (< (stri i x86-64) *2^48*))
   :rule-classes :linear))

; SSR read lemmas

(defthm ssrp-forward
  (implies (ssrp x)
           (nat-listp x))
  :rule-classes :forward-chaining)

(defthm natp-ssri
  (implies (forced-and (x86-64p x86-64)
                       (n01p i))
           (and (integerp (ssri i x86-64))
                (<= 0 (ssri i x86-64))))
  :rule-classes :type-prescription)

(encapsulate
 ()

 (local (defthm nth-of-ssr-<=expt-2-16
          (implies (and (ssrp x)
                        (integerp i)
                        (<= 0 i)
                        (< i (len x)))
                   (< (nth i x) *2^16*))
          :rule-classes :linear
          :hints (("Goal" :in-theory (e/d (nth) ())))))

 (defthm ssri-less-than-expt-2-16
   (implies (forced-and (x86-64p x86-64)
                        (n01p i))
            (< (ssri i x86-64) *2^16*))
   :rule-classes :linear))
```

```
; CTR read lemmas

(defthm ctrp-forward
  (implies (ctrp x)
           (nat-listp x))
  :rule-classes :forward-chaining)

(defthm natp-ctri
  (implies (forced-and (x86-64p x86-64)
                       (natp i)
                       (< i 6))
           (and (integerp (ctri i x86-64))
                (<= 0 (ctri i x86-64))))
  :rule-classes :type-prescription)

(encapsulate
 ()

 (local (defthm nth-of-ctr-<=expt-2-64
          (implies (and (ctrp x)
                        (integerp i)
                        (<= 0 i)
                        (< i (len x)))
                   (< (nth i x) *2^64*))
          :rule-classes :linear
          :hints (("Goal" :in-theory (e/d (nth) ())))))

 (defthm ctri-less-than-expt-2-64
   (implies (forced-and (x86-64p x86-64)
                        (natp i)
                        (< i 6))
            (< (ctri i x86-64) *2^64*))
   :rule-classes :linear))

; MEM-ARRAY read lemmas

(defthm mem-arrayp-forward
  (implies (mem-arrayp x)
           (nat-listp x))
  :rule-classes :forward-chaining)

(defthm natp-mem-arrayp-when-valid-mem-array-index
  (implies (forced-and (x86-64p-pre x86-64)
                       (natp i)
                       (< i (mem-array-length x86-64)))
           (and (integerp (mem-arrayi i x86-64))
                (<= 0 (mem-arrayi i x86-64))))
  :rule-classes (:rewrite :type-prescription))

(encapsulate
 ()
 (local (defthm nth-of-mem-array-<=-expt-2-64
          (implies (and (mem-arrayp x)
                        (integerp i)
                        (<= 0 i)
                        (< i (len x)))
                   (< (nth i x) *2^64*))
          :rule-classes :linear
          :hints (("Goal" :in-theory (e/d (nth) ())))))

 (defthm mem-arrayi-less-than-expt-2-64
   (implies (forced-and (x86-64p-pre x86-64)
                        (natp i)
                        (< i (mem-array-length x86-64)))
            (< (mem-arrayi i x86-64) *2^64*))
   :rule-classes :linear))

#|| Relics of 32-bit model -- might need to be restored suitably.

; SEG-BASE read lemmas

(defthm seg-basep-forward
  (implies (seg-basep x)
           (nat-listp x))
  :rule-classes :forward-chaining)

(defthm natp-seg-basei
  (implies (forced-and (x86-64p x86-64)
                       (natp i)
                       (< i 6))
           (and (integerp (seg-basei i x86-64))
                (<= 0 (seg-basei i x86-64))))
  :rule-classes :type-prescription)

(encapsulate
 ()

 (local (defthm nth-of-seg-base-<=expt-2-32
          (implies (and (seg-basep x)
                        (integerp i)
                        (<= 0 i)
                        (< i (len x)))
                   (< (nth i x) *2^32*))
          :rule-classes :linear
          :hints (("Goal" :in-theory (e/d (nth) ())))))

 (defthm seg-basei-less-than-expt-2-32
   (implies (forced-and (x86-64p x86-64)
                        (natp i)
                        (< i 6))
            (< (seg-basei i x86-64) *2^32*))
   :rule-classes :linear))

; SEG-LIMIT read lemmas

(defthm seg-limitp-forward
  (implies (seg-limitp x)
           (nat-listp x))
  :rule-classes :forward-chaining)

(defthm natp-seg-limiti
  (implies (forced-and (x86-64p x86-64)
                       (natp i)
                       (< i 6))
           (and (integerp (seg-limiti i x86-64))
                (<= 0 (seg-limiti i x86-64))))
  :rule-classes :type-prescription)

(encapsulate
 ()

 (local (defthm nth-of-seg-limit-<=1048576
          (implies (and (seg-limitp x)
                        (integerp i)
                        (<= 0 i)
                        (< i (len x)))
                   (< (nth i x) 1048576))
          :rule-classes :linear
          :hints (("Goal" :in-theory (e/d (nth) ())))))

 (defthm seg-limiti-less-than-expt-2-64
   (implies (forced-and (x86-64p x86-64)
                        (natp i)
                        (< i 6))
            (< (seg-limiti i x86-64) 1048576))
   :rule-classes :linear))

; SEG-ACCESS read lemmas

(defthm seg-accessp-forward
  (implies (seg-accessp x)
           (nat-listp x))
  :rule-classes :forward-chaining)

(defthm natp-seg-accessi
  (implies (forced-and (x86-64p x86-64)
                       (natp i)
                       (< i 6))
           (and (integerp (seg-accessi i x86-64))
                (<= 0 (seg-accessi i x86-64))))
  :rule-classes :type-prescription)
```

```
(encapsulate
 ()

 (local (defthm nth-of-seg-access-<=4096
          (implies (and (seg-accessp x)
                        (integerp i)
                        (<= 0 i)
                        (< i (len x)))
                   (< (nth i x) 4096))
          :rule-classes :linear
          :hints (("Goal" :in-theory (e/d (nth) ())))))

 (defthm seg-accessi-less-than-expt-2-64
   (implies (forced-and (x86-64p x86-64)
                        (natp i)
                        (< i 6))
            (< (seg-accessi i x86-64) 4096))
   :rule-classes :linear))

||#

; We wonder if the two lemmas about !xxx would be better as
; :FORWARD-CHAINING rules (or, as both :REWRITE and :FORWARD-CHAINING
; rules), using *MEM-SIZE-IN-BYTES* and *REG-SIZE-IN-DWRDS* in the
; hypotheses instead of LEN.

(defthm length-is-len-when-not-stringp

; This lemma is important because we are keeping length disabled.  An
; alternative may be to keep length enabled; then a case split on (stringp x)
; would provide a clue that we are missing some hypothesis or fact.

  (implies (not (stringp x))
           (equal (length x)
                  (len x)))
  :hints (("Goal" :in-theory (e/d (length) ()))))

; RGF update lemmas

(defthm rgfp-update-nth
  (implies (forced-and (rgfp x)
                       (integerp i)
                       (<= 0 i)
                       (< i (len x))
                       (n64p v))
           (rgfp (update-nth i v x))))

(defthm expected-mem-array-next-addr-only-depends-on-mem-table-lemma
  (implies (equal (nth *mem-tablei* x86-64-alt)
                  (nth *mem-tablei* x86-64))
           (equal (expected-mem-array-next-addr i j x86-64-alt)
                  (expected-mem-array-next-addr i j x86-64)))
  :hints (("Goal" :in-theory (enable expected-mem-array-next-addr)))
  :rule-classes nil)

(defthm expected-mem-array-next-addr-only-depends-on-mem-table
  (implies (and (equal (expected-mem-array-next-addr 0 *2^27* x86-64)
                       x) ; free var
                (syntaxp (equal x86-64 'x86-64))
                (equal (nth *mem-tablei* x86-64-alt)
                       (nth *mem-tablei* x86-64)))
           (equal (expected-mem-array-next-addr 0 134217728 x86-64-alt)
                  x))
  :hints (("Goal"
           :use
           ((:instance
             expected-mem-array-next-addr-only-depends-on-mem-table-lemma
             (i 0) (j *2^27*))))))

(defthm x86-64p-!rgfi-n04p
  (implies (forced-and (x86-64p x86-64)
                       (n04p i)
                       (n64p v))
           (x86-64p (!rgfi i v x86-64)))
  :hints (("Goal" ; gross hint
           :expand ((nth *mem-tablei* x86-64)))))

; RIP update lemma

(defthm x86-64p-!rip
  (implies (forced-and (x86-64p x86-64)
                       (n48p v))
           (x86-64p (!rip v x86-64))))

; EFLAGS update lemma

(defthm x86-64p-!flg
  (implies (forced-and (x86-64p x86-64)
                       (n64p v))
           (x86-64p (!flg v x86-64))))

; SEG update lemmas

(defthm segp-update-nth
  (implies (forced-and (segp x)
                       (integerp i)
                       (<= 0 i)
                       (< i (len x))
                       (n16p v))
           (segp (update-nth i v x))))

(defthm x86-64p-!segi
  (implies (forced-and (x86-64p x86-64)
                       (n01p i)
                       (n16p v))
           (x86-64p (!segi i v x86-64))))

; STR update lemmas

(defthm strp-update-nth
  (implies (forced-and (strp x)
                       (integerp i)
                       (<= 0 i)
                       (< i (len x))
                       (n48p v))
           (strp (update-nth i v x))))

(defthm x86-64p-!stri
  (implies (forced-and (x86-64p x86-64)
                       (n01p i)
                       (n48p v))
           (x86-64p (!stri i v x86-64))))

; SSR update lemmas

(defthm ssrp-update-nth
  (implies (forced-and (ssrp x)
                       (integerp i)
                       (<= 0 i)
                       (< i (len x))
                       (n16p v))
           (ssrp (update-nth i v x))))

(defthm x86-64p-!ssri
  (implies (forced-and (x86-64p x86-64)
                       (n01p i)
                       (n16p v))
           (x86-64p (!ssri i v x86-64))))

; CTR update lemmas

(defthm ctrp-update-nth
  (implies (forced-and (ctrp x)
                       (integerp i)
                       (<= 0 i)
                       (< i (len x))
                       (n64p v))
           (ctrp (update-nth i v x))))

(defthm x86-64p-!ctri
  (implies (forced-and (x86-64p x86-64)
                       (integerp i)
                       (<= 0 i)
```

```
                      (< i 6)
                      (n64p v))
           (x86-64p (!ctri i v x86-64))))

; Memory access definitions

; Start admission of memi

(encapsulate
 ()

 (local (include-book "arithmetic-5/top" :dir :system))

 (defthm n45p-ash--18-is-n27p
   (implies (n45p x)
            (n27p (ash x -18)))
   :rule-classes ((:type-prescription
                   :corollary (implies (n45p x)
                                       (natp (ash x -18))))
                  (:linear
                   :corollary (implies (n45p x)
                                       (< (ash x -18) *2^27*))))))

; It is convenient to separate out the mem-table from the rest of the stobj, so
; that we don't have to reason about the mem-table after updating some other
; stobj field (see e.g. x86-64p-!rgfi-n04p).

(defthm good-mem-table-entriesp-logic-property
  (let ((addr (nth i mem-table)))
    (implies (and (good-mem-table-entriesp-logic
                   z table-max-index array-next-addr mem-table)
                  (natp i)
                  (<= z i)
                  (<= i table-max-index)
                  (not (eql addr 1)))
             (and (natp addr)
                  (equal (logand #x3ffff addr) 0)
                  (< (+ #x3ffff addr) array-next-addr))))
  :hints (("Goal"
           :in-theory (enable good-mem-table-entriesp-logic)
           :induct (good-mem-table-entriesp-logic
                    z table-max-index array-next-addr mem-table))))

; The following requires properties of good-memp.
(defthm mem-array-next-addr-lte-mem-array-length
  (implies (x86-64p x86-64)
           (<= (nth *mem-array-next-addr* x86-64)
               (len (nth *mem-arrayi* x86-64))))
  :rule-classes :linear)

(defthm mem-array-next-addr-natp
  (implies (x86-64p x86-64)
           (natp (nth *mem-array-next-addr* x86-64)))
  :rule-classes :type-prescription)

(defthm x86-64p-forward-to-x86-64p-pre
  (implies (x86-64p x86-64)
           (x86-64p-pre x86-64))
  :rule-classes :forward-chaining)

(defthm x86-64p-forward-to-good-memp
  (implies (x86-64p x86-64)
           (good-memp x86-64))
  :rule-classes :forward-chaining)

; The conclusions of lemmas such as the following are about nth instead of
; about (mem-tablei i x86-64).  This way, we can leave functions like
; mem-tablei enabled when doing low-level reasoning, which is great because
; then we can rely on nth-update-nth to do our simplification, rather than
; having to prove n^2 rules such as (rip (update-memi ... x86-64)) = (rip
; x86-64).

(defthm natp-mem-tablei
  (implies (forced-and (x86-64p-pre x86-64)
                       (natp i)
                       (< i *mem-table-size*))
           (natp (nth i (nth *mem-tablei* x86-64))))
  :rule-classes :type-prescription)

; The following requires properties of good-memp.
(defthm logand-mem-tablei-is-0
  (let ((addr (nth i (nth *mem-tablei* x86-64))))
    (implies (and (force (x86-64p x86-64))
                  (force (natp i))
                  (force (< i *mem-table-size*))
                  (not (eql addr 1)))
             (equal (logand #x3ffff addr) 0))))

; The following requires properties of good-memp.
(defthm good-memp-linear-1
  (let ((addr (nth i (nth *mem-tablei* x86-64))))
    (implies (and (force (x86-64p x86-64))
                  (force (natp i))
                  (force (< i *mem-table-size*))
                  (not (eql addr 1)))
             (< (+ #x3ffff addr) (nth *mem-array-next-addr* x86-64))))
  :rule-classes :linear)

; The following requires properties of good-memp.
(defthm good-memp-linear-2

; This lemma originally had hypothesis (not (eql addr 1)) at the end.
; That proved to be very expensive for the proof of
; x86-64p-pre-update-mem-array-next-addr-+, I'm guessing because x86-64p was
; enabled.

  (let ((addr (nth i (nth *mem-tablei* x86-64))))
    (implies (and (not (eql addr 1))
                  (force (x86-64p x86-64))
                  (force (natp i))
                  (force (< i *mem-table-size*)))
             (< (+ #x3ffff addr) (len (nth *mem-arrayi* x86-64)))))
  :rule-classes :linear)

(in-theory (disable x86-64p x86-64p-pre))

(defun memi (i x86-64)
  (declare (xargs :stobjs x86-64
                  :guard (and (n45p i) ; quadword address
                              (x86-64p x86-64))))
  (let* ((i-top27 (ash i -18))
         (addr (mem-tablei i-top27 x86-64)))
    (if (eql addr 1) ; page is not present
        *default-mem-value*
      (let ((index (logior addr (logand #x3ffff i))))
        (mem-arrayi index x86-64)))))

; Start proof obligations for guard verification for add-page.

(defthmd mem-array-next-addr-is-expected-mem-array-next-addr
  (implies (x86-64p x86-64)
           (equal (nth *mem-array-next-addr* x86-64)
                  (expected-mem-array-next-addr 0
                                                (mem-table-length x86-64)
                                                x86-64)))
  :hints (("Goal" :in-theory (enable x86-64p))))

(encapsulate
 ()

 (local (include-book "arithmetic-5/top" :dir :system))

 (defthm expected-mem-array-next-addr-bound-general
   (implies (and (equal 1 (nth j (nth *mem-tablei* x86-64)))
                 (natp i)
                 (natp j)
                 (natp k)
                 (<= i j)
                 (< j k)
                 (<= k *2^27*))
            (<= (expected-mem-array-next-addr i k x86-64)
                (* *2^18* (1- (- k i)))))
   :hints (("Goal" :expand ((expected-mem-array-next-addr j k x86-64))
            :use (expected-mem-array-in-parts)))
   :rule-classes nil))

(defthm expected-mem-array-next-addr-bound-linear
   (implies (and (equal 1 (nth i (nth *mem-tablei* x86-64))) ; i is free
                 (force (natp i))
                 (force (< i *2^27*)))
            (<= (expected-mem-array-next-addr 0 *2^27* x86-64)
                (- *2^45* *2^18*)))
   :hints (("Goal" ;:expand ((expected-mem-array-next-addr j k x86-64))
            :use ((:instance expected-mem-array-next-addr-bound-general
                             (i 0)
                             (j i)
                             (k *2^27*)))))
   :rule-classes :linear)

(defthm mem-array-next-addr-increment-bound
  (implies (and (equal 1 (nth i (nth *mem-tablei* x86-64))) ; i is free
                (force (x86-64p x86-64))
                (force (natp i))
                (force (< i *2^27*)))
           (<= (+ *2^18* (nth *mem-array-next-addr* x86-64))
               *2^45*))
  :hints (("Goal"
           :in-theory
           (enable mem-array-next-addr-is-expected-mem-array-next-addr)))
  :rule-classes :linear)

(defun add-page (i x86-64)
  (declare (xargs :stobjs x86-64
                  :guard (and (n27p i) ; index into mem-table
                              (x86-64p x86-64)
                              (equal 1 ; "page not present"
                                     (mem-tablei i x86-64)))))
  (let* ((addr (mem-array-next-addr x86-64))
         (len (mem-array-length x86-64))
         (x86-64 (cond ((eql addr len) ; must resize!
                        (resize-mem-array (min (* *mem-array-resize-factor*
                                                  len)
                                               *2^45*)
                                          x86-64))
                       (t x86-64)))
         (x86-64 (!mem-array-next-addr (+ addr *2^18*) x86-64))
         (x86-64 (!mem-tablei i addr x86-64)))
    (mv addr x86-64)))

; Start guard proof for !memi.

(defthm len-resize-list
  (equal (len (resize-list lst n default-value))
         (nfix n)))
```

```
(encapsulate
 ()
 (local (include-book "arithmetic-5/top" :dir :system))

 (defthm logand-expected-mem-array-next-addr
   (equal (logand #x3ffff
                  (expected-mem-array-next-addr i j x86-64))
          0)
   :hints (("Goal" :in-theory (enable expected-mem-array-next-addr))))

 (defthm *-3-preserves-0-mod-2^18
   (implies (and (natp x)
                 (equal (logand #x3ffff x) 0))
            (equal (logand #x3ffff (* 3 x)) 0))))

(encapsulate
 ()

 (local (defthm logior-logand-2^18-3
          (implies (and (natp x) (natp y))
                   (<= (logior x (logand #x3ffff y))
                       (logior #x3ffff x)))
          :rule-classes nil))

 (local (include-book "arithmetic-5/top" :dir :system))
 (local (include-book "rtl/rel8/lib/top" :dir :system))

 (defthm logior-logand-2^18
   (implies (and (natp x)
                 (natp y)
                 (natp z)
                 (< x z)
                 (equal (logand #x3ffff x) 0)
                 (equal (logand #x3ffff z) 0))
            (< (logior x
                       (logand #x3ffff y))
               z))
   :hints (("Goal" :use ((:instance logior-logand-2^18-1 (x x))
                         (:instance logior-logand-2^18-1 (x z))
                         (:instance logior-logand-2^18-2
                                    (x1 (floor x *2^18*))
                                    (y (logand #x3ffff y)))
                         logior-logand-2^18-3)
            :in-theory (disable logior-logand-inequality
                                logior-logand-2^18-2
                                logand-constant-mask
                                ;; avoid loop:
                                prefer-positive-addends-<
                                a2
                                reduce-integerp-+)))))
```

```
(encapsulate
 ()

 (local (include-book "arithmetic-5/top" :dir :system))

 (defthm <-preserved-by-adding-<-2^18
   (implies (and (< next len)
                 (n18p i)
                 (force (natp next))
                 (force (natp len))
                 (force (equal (logand #x3ffff next) 0))
                 (equal (logand #x3ffff len) 0))
            (< (+ next i)
               len))))

(defun !memi (i v x86-64)
  (declare (xargs :stobjs x86-64
                  :guard (and (n45p i)
                              (n64p v)
                              (x86-64p x86-64))
                  :guard-hints (("Goal" :in-theory (enable x86-64p)))))
  (let* ((i-top27 (ash i -18))
         (addr (mem-tablei i-top27 x86-64)))
    (mv-let (addr x86-64)
            (cond ((eql addr 1) ; page is not present
                   (add-page i-top27 x86-64))
                  (t (mv addr x86-64)))
            (!mem-arrayi (logior addr (logand #x3ffff i))
                         v
                         x86-64))))

(in-theory (disable memi !memi))

; Memory update lemmas

; Start proof of x86-64p-!memi

; Start proof of x86-64p-pre-!memi

(defthm mem-arrayp-update-nth
  (implies (and (force (< i (len mem-array)))
                (force (n64p v))
                (force (mem-arrayp mem-array)))
           (mem-arrayp (update-nth i v mem-array))))

(defthm mem-tablep-update-nth
  (implies (and (force (< i (len mem-table)))
                (force (n45p v))
                (force (mem-tablep mem-table)))
           (mem-tablep (update-nth i v mem-table))))
```

```
; Start proof of x86-64p-!memi

; Start proof of x86-64p-!memi-not-new-page

; Start proof of x86-64p-!mem-arrayi

(defthm x86-64p-pre-update-nth-mem-arrayi
  (implies (forced-and (x86-64p-pre x86-64)
                       (n64p v)
                       (natp addr)
                       (< addr (len (nth *mem-arrayi* x86-64))))
           (x86-64p-pre (update-nth
                         *mem-arrayi*
                         (update-nth addr v (nth *mem-arrayi* x86-64))
                         x86-64)))
  :hints (("Goal" :in-theory (enable x86-64p-pre))))

(defthm good-mem-arrayp-1-logic-update-nth
  (implies (and (n64p v)
                (natp addr)
                (< addr mem-array-next-addr)
                (good-mem-arrayp-1-logic mem-array-next-addr
                                         (len mem-array)
                                         mem-array))
           (good-mem-arrayp-1-logic mem-array-next-addr
                                    (len mem-array)
                                    (update-nth addr v mem-array)))
  :hints (("Goal" :in-theory (enable x86-64p))
          ("[1]Goal" :in-theory (enable x86-64p))))

(defthm x86-64p-update-nth-mem-arrayi
  (implies (forced-and (x86-64p x86-64)
                       (n64p v)
                       (natp addr)
                       (< addr (nth *mem-array-next-addr* x86-64)))
           (x86-64p (update-nth
                     *mem-arrayi*
                     (update-nth addr v (nth *mem-arrayi* x86-64))
                     x86-64)))
  :hints (("Goal" :in-theory (enable x86-64p))
          ("[1]Goal" :in-theory (enable x86-64p))))

(defthm x86-64p-!memi-not-new-page
  (implies (and (x86-64p x86-64)
                (n45p i)
                (n64p v)
                (not (equal (mem-tablei (ash i -18) x86-64)
                            1)))
           (x86-64p (!mem-arrayi (logior (nth (ash i -18)
                                              (nth *mem-tablei* x86-64))
```

```
                                  (logand #x3ffff i))
                            v
                            x86-64)))
  :rule-classes nil)

; Start proof of x86-64p-!memi-new-page

; Start proof of x86-64p-!memi-new-page-no-resize

(defthm add-2^18-preserves-<=
  (implies (and (< x bound)
                (force (natp bound))
                (equal (logand #x3ffff bound) 0)
                (force (natp x))
                (force (equal (logand #x3ffff x) 0)))
           (<= (+ *2^18* x) bound))
  :hints (("Goal" :use ((:instance logior-logand-2^18
                                   (x x)
                                   (y #x3ffff)
                                   (z bound))))))

(defthm expected-mem-array-next-addr-update-nth-mem-tablei
  (implies (forced-and (equal (nth index (nth *mem-tablei* x86-64))
                              1)
                       (natp index)
                       (natp lower)
                       (natp upper)
                       (<= upper *2^27*)
                       (not (equal any-value 1))
                       (equal mem-table (nth *mem-tablei* x86-64)))
           (equal (expected-mem-array-next-addr
                   lower upper
                   (update-nth *mem-tablei*
                               (update-nth index any-value mem-table)
                               x86-64))
                  (if (and (<= lower index)
                           (< index upper))
                      (+ *2^18*
                         (expected-mem-array-next-addr
                          lower upper
                          x86-64))
                    (expected-mem-array-next-addr
                     lower upper
                     x86-64)))))
  :hints (("Goal" :in-theory (enable expected-mem-array-next-addr))))

(defthm logand-mem-array-next-addr
  (implies (good-memp x86-64)
           (equal (logand #x3ffff
                          (nth *mem-array-next-addr* x86-64))
                  0)))
```

```
(defthm logand-3ffff-0-implies-not-1
  (implies (equal (logand #x3ffff x) 0)
           (not (equal x 1))))

(defthm good-mem-table-entriesp-logic-update
  (implies
   (and (natp index)
        (good-mem-table-entriesp-logic lower upper array-next-addr mem-table)
        (natp val)
        (<= val array-next-addr)
        (equal (logand #x3ffff val) 0))
   (good-mem-table-entriesp-logic
    lower upper
    (+ 262144 array-next-addr)
    (update-nth index val mem-table)))
  :hints (("Goal" :in-theory (enable nth good-mem-table-entriesp-logic))))

; Start proof of good-mem-table-no-dupsp-logic-update-nth
; (and good-mem-table-no-dupsp-logic-update-nth-lemma).

(defthm member-revappend-lemma
  (implies (member a y)
           (member a (revappend x y))))

(defthm member-revappend
  (iff (member a (revappend x y))
       (or (member a x)
           (member a y))))

(defthm member-merge-<-into->
  (iff (member a (merge-<-into-> x y z))
       (or (member a x)
           (member a y)
           (member a z))))

(defthm member-merge->-into-<
  (iff (member a (merge->-into-< x y z))
       (or (member a x)
           (member a y)
           (member a z))))

(defthm no-duplicatesp-sorted-revappend-2
  (implies (not (no-duplicatesp-sorted y))
           (not (no-duplicatesp-sorted (revappend x y)))))

(defthm no-duplicatesp-sorted-revappend-1
  (implies (not (no-duplicatesp-sorted x))
           (not (no-duplicatesp-sorted (revappend x y)))))

(defthm no-duplicatesp-sorted-merge-<-into->-3
  (implies (not (no-duplicatesp-sorted z))
           (not (no-duplicatesp-sorted (merge-<-into-> x y z)))))

(defthm no-duplicatesp-sorted-merge-<-into->-1
  (implies (not (no-duplicatesp-sorted x))
           (not (no-duplicatesp-sorted (merge-<-into-> x y z)))))

(defthm no-duplicatesp-sorted-merge-<-into->-2
  (implies (not (no-duplicatesp-sorted y))
           (not (no-duplicatesp-sorted (merge-<-into-> x y z)))))

(defthm no-duplicatesp-sorted-merge->-into-<-3
  (implies (not (no-duplicatesp-sorted z))
           (not (no-duplicatesp-sorted (merge->-into-< x y z)))))

(defthm no-duplicatesp-sorted-merge->-into-<-1
  (implies (not (no-duplicatesp-sorted x))
           (not (no-duplicatesp-sorted (merge->-into-< x y z)))))

(defthm no-duplicatesp-sorted-merge->-into-<-2
  (implies (not (no-duplicatesp-sorted y))
           (not (no-duplicatesp-sorted (merge->-into-< x y z)))))

(defun all-< (lst x)
  (cond ((endp lst) t)
        ((< (car lst) x)
         (all-< (cdr lst) x))
        (t nil)))

(defthm member-implies-not-all-<
  (implies (and (not (< b a))
                (member b y))
           (not (all-< y a))))

(defthm all-<-revappend
  (equal (all-< (revappend x y) a)
         (and (all-< x a)
              (all-< y a))))

(defthm all-<-merge-<-into->
  (equal (all-< (merge-<-into-> x y z) a)
         (and (all-< x a)
              (all-< y a)
              (all-< z a))))

(defthm all-<-merge->-into-<
  (equal (all-< (merge->-into-< x y z) a)
         (and (all-< x a)
              (all-< y a)
              (all-< z a))))
```

```
; Start proof of all-<-mem-table-entries-logic-next-addr

(encapsulate
 ()

 (local (in-theory (enable good-mem-table-entriesp-logic)))

 (local (defthm good-mem-table-entriesp-logic-preserved-lemma
          (implies (and (good-mem-table-entriesp-logic lower upper1 array-bound
                                                       x86-64)
                        (natp upper2)
                        (<= lower upper2)
                        (<= upper2 upper1))
                   (good-mem-table-entriesp-logic lower upper2 array-bound
                                                  x86-64))
          :hints (("Goal" :in-theory (disable (force))))))

 (defthm good-mem-table-entriesp-logic-preserved
   (implies (and (good-mem-table-entriesp-logic lower1 upper1 array-bound
                                                x86-64)
                 (natp lower2)
                 (natp upper2)
                 (<= lower1 lower2)
                 (<= lower2 upper2)
                 (<= upper2 upper1))
            (good-mem-table-entriesp-logic lower2 upper2 array-bound x86-64))
   :hints (("Goal" :in-theory (disable (force))))))

(defthm all-<-mem-table-entries-logic-next-addr
  (implies (good-mem-table-entriesp-logic lower upper next-addr mem-table)
           (all-< (mem-table-entries-logic lower upper mem-table parity)
                  next-addr))
  :hints (("Goal" :expand ((good-mem-table-entriesp-logic lower (+ 1 lower)
                                                          next-addr mem-table)
                           (good-mem-table-entriesp-logic (+ 1 lower)
                                                          (+ 1 lower)
                                                          next-addr
                                                          mem-table)
                           (good-mem-table-entriesp-logic lower lower next-addr
                                                          mem-table)))))

(defthm merge-<-into->-append-1
  (implies (and (all-< x a)
                (all-< y a))
           (equal (merge-<-into-> (append x (list a)) y z)
                  (cons a (merge-<-into-> x y z)))))

(defthm merge-<-into->-append-2
  (implies (and (all-< x a)
                (all-< y a))
           (equal (merge-<-into-> x (append y (list a)) z)
                  (cons a (merge-<-into-> x y z)))))

(defthm merge->-into-<-append-last
  (equal (merge->-into-< x y (append z1 z2))
         (append (merge->-into-< x y z1)
                 z2))
  :rule-classes nil)

(defthm merge->-into-<-append-1
  (implies (and (all-< x a)
                (all-< y a))
           (equal (merge->-into-< (cons a x) y nil)
                  (append (merge->-into-< x y nil)
                          (list a))))
  :hints (("Goal" :use ((:instance merge->-into-<-append-last
                                   (x x)
                                   (y y)
                                   (z1 nil)
                                   (z2 (list a))))
           :expand ((merge->-into-< (cons a x) y nil)))))

(defthm merge->-into-<-append-2
  (implies (and (all-< x a)
                (all-< y a))
           (equal (merge->-into-< x (cons a y) nil)
                  (append (merge->-into-< x y nil)
                          (list a))))
  :hints (("Goal" :use ((:instance merge->-into-<-append-last
                                   (x x)
                                   (y y)
                                   (z1 nil)
                                   (z2 (list a))))
           :expand ((merge->-into-< x (cons a y) nil)))))

(defthm mem-table-entries-logic-update-nth
  (implies (and (natp next-addr)
                (not (equal next-addr 1))
                (natp lower)
                (natp upper)
                (natp index)
                (< index (len mem-table))
                (equal (nth index mem-table) 1)
                (booleanp parity)
                (all-< (mem-table-entries-logic lower upper mem-table parity)
                       next-addr))
           (equal (mem-table-entries-logic lower upper
                                           (update-nth index next-addr mem-table)
                                           parity)
                  (if (and (<= lower index)
                           (<= index upper))
                      (if parity
                         (append (mem-table-entries-logic lower upper mem-table
                                                          parity)
                                 (list next-addr))
                       (cons next-addr
                             (mem-table-entries-logic lower upper mem-table
                                                      parity)))
                   (mem-table-entries-logic lower upper mem-table parity)))))

(defthm not-all-<-car
  (equal (all-< x (car x))
         (not (consp x))))

(defthm no-duplicatesp-sorted-append
  (equal (no-duplicatesp-sorted (append x y))
         (and (no-duplicatesp-sorted x)
              (no-duplicatesp-sorted y)
              (or (atom x)
                  (atom y)
                  (not (equal (car (last x)) (car y)))))))

(defthm not-all-<-car-last
  (equal (all-< x (car (last x)))
         (not (consp x))))

(defthm good-mem-table-no-dupsp-logic-update-nth-lemma-0
  (implies (and (natp lower)
                (natp upper)
                (natp index)
                (< index (len mem-table))
                (natp next-addr)
                (not (equal next-addr 1))
                (booleanp parity)
                (equal (nth index mem-table) 1)
                (no-duplicatesp-sorted
                 (mem-table-entries-logic lower upper mem-table parity))
                (all-< (mem-table-entries-logic lower upper mem-table parity)
                       next-addr))
           (no-duplicatesp-sorted
            (mem-table-entries-logic lower upper
                                     (update-nth index next-addr mem-table)
                                     parity))))

(in-theory (disable mem-table-entries-logic-update-nth))

(defthm good-mem-table-no-dupsp-logic-update-nth-lemma
  (implies (and (natp lower)
                (natp upper)
                (natp index)
                (< index (len mem-table))
                (<= lower index)
                (<= index upper)
                (natp next-addr)
                (not (equal 1 next-addr))
                (booleanp parity)
                (equal (nth index mem-table) 1)
                (no-duplicatesp-sorted
                 (mem-table-entries-logic lower upper mem-table parity))
                (good-mem-table-entriesp-logic lower upper next-addr
                                               mem-table))
           (no-duplicatesp-sorted
            (mem-table-entries-logic lower upper
                                     (update-nth index next-addr mem-table)
                                     parity))))

(defthm good-mem-table-no-dupsp-logic-update-nth
  (implies (and (natp lower)
                (natp upper)
                (natp index)
                (< index (len mem-table))
                (<= lower index)
                (<= index upper)
                (natp next-addr)
                (not (equal 1 next-addr))
                (equal (nth index mem-table) 1)
                (good-mem-table-no-dupsp-logic lower upper mem-table)
                (good-mem-table-entriesp-logic lower upper next-addr
                                               mem-table))
           (good-mem-table-no-dupsp-logic lower upper
                                          (update-nth index next-addr
                                                      mem-table)))
  :hints (("Goal" :in-theory (enable good-mem-table-no-dupsp-logic
                                     good-mem-table-entriesp-logic))))

(defthm good-mem-arrayp-1-logic-preserved-upward
  (implies (and (good-mem-arrayp-1-logic index1 len mem-array)
                (natp index1)
                (natp index2)
                (<= index1 index2))
           (good-mem-arrayp-1-logic index2 len mem-array)))

(defthm x86-64p-!memi-new-page-no-resize-lemma-1
  (implies
   (and (x86-64p x86-64)
        (n27p index) ; (ash i -18)
        (equal (mem-tablei index x86-64)
               1)
        (not (equal (nth *mem-array-next-addr* x86-64)
                    (len (nth *mem-arrayi* x86-64)))))
   (x86-64p
    (update-nth *mem-tablei*
                (update-nth index
                            (nth *mem-array-next-addr* x86-64)
```

```
                           (nth *mem-tablei* x86-64))
                (update-nth *mem-array-next-addr*
                           (+ *2^18*
                              (nth *mem-array-next-addr* x86-64))
                           x86-64))))
  :hints (("Goal"
           :in-theory
           (enable mem-array-next-addr-is-expected-mem-array-next-addr
                   x86-64p x86-64p-pre))))

(defthm x86-64p-!memi-new-page-no-resize-lemma-2
  (implies (and (x86-64p x86-64)
                (n64p v)
                (natp addr)
                (equal (logand #x3ffff addr) 0)
                (< ADDR (NTH *MEM-ARRAY-NEXT-ADDR* X86-64))
                ;(< addr (len mem-array))
                (equal mem-array (nth *mem-arrayi* x86-64)))
           (x86-64p
            (update-nth
             *mem-arrayi*
             (update-nth (+ addr (logand #x3ffff i))
                         v
                         mem-array)
             x86-64))))

(defthm x86-64p-!memi-new-page-no-resize
  (implies
   (and (x86-64p x86-64)
        (n45p i)
        (n64p v)
        (equal (mem-tablei (ash i -18) x86-64)
               1)
        (not (equal (nth *mem-array-next-addr* x86-64)
                    (len (nth *mem-arrayi* x86-64)))))
   (x86-64p
    (update-nth
     *mem-arrayi*
     (update-nth (logior (nth *mem-array-next-addr* x86-64)
                         (logand #x3ffff i))
                 v (nth *mem-arrayi* x86-64))
     (update-nth *mem-tablei*
                 (update-nth (ash i -18)
                             (nth *mem-array-next-addr* x86-64)
                             (nth *mem-tablei* x86-64))
                 (update-nth *mem-array-next-addr*
                             (+ *2^18*
                                (nth *mem-array-next-addr* x86-64))
                             x86-64)))))
  :rule-classes nil)
```

```
; Start proof of x86-64p-!memi-new-page-resize

(defthm mem-arrayp-resize-list
  (implies (and (mem-arrayp lst)
                (unsigned-byte-p 64 default-value))
           (mem-arrayp (resize-list lst new-len default-value))))

(defun nth-resize-list-induction (i lst n default-value)
  (declare (ignorable i lst n default-value))
  (if (posp n)
      (nth-resize-list-induction (1- i)
                                 (if (atom lst) lst (cdr lst))
                                 (1- n)
                                 default-value)
    nil))

(defthm nth-resize-list
  (implies (and (natp i)
                (natp n)
                (<= (len lst) n)
                (< i n))
           (equal (nth i (resize-list lst n default))
                  (if (< i (len lst))
                      (nth i lst)
                    default)))
  :hints (("Goal" :in-theory (enable resize-list nth)
           :induct (nth-resize-list-induction i lst n default-value))))

(defthm good-mem-arrayp-1-logic-resize-list
  (implies (and (natp next-addr)
                (natp new-len)
                (<= (len mem-array) new-len)
                (good-mem-arrayp-1-logic next-addr
                                         (len mem-array)
                                         mem-array))
           (good-mem-arrayp-1-logic next-addr
                                    new-len
                                    (resize-list mem-array new-len 0)))
; It's very unusual that the following works but :hints (("Goal" :induct t))
; does not!
  :instructions ((then induct prove)))

(defthm x86-64p-resize-mem-array
   (implies
    (and (x86-64p x86-64)
         (natp new-len)
         (equal (logand #x3ffff new-len) 0)
         (< (len (nth *mem-arrayi* x86-64)) new-len)
         (<= new-len *2^45*))
    (x86-64p (update-nth *mem-arrayi*
                         (resize-list (nth *mem-arrayi* x86-64)
                                    new-len 0)
                          x86-64)))
   :hints (("Goal" :in-theory (enable x86-64p x86-64p-pre))))

(defthm x86-64p-!memi-new-page-resize
  (implies
   (and (x86-64p x86-64)
        (n45p i)
        (n64p v)
        (equal (mem-tablei (ash i -18) x86-64)
               1)
        (equal (nth *mem-array-next-addr* x86-64)
               (len (nth *mem-arrayi* x86-64)))
        (natp new-len)
        (equal (logand #x3ffff new-len) 0)
        (< (len (nth *mem-arrayi* x86-64)) new-len)
        (<= new-len *2^45*))
   (x86-64p
    (update-nth
     *mem-arrayi*
     (update-nth (+ (nth *mem-array-next-addr* x86-64)
                    (logand #x3ffff i))
                 v
                 (resize-list (nth *mem-arrayi* x86-64)
                              new-len
                              0))
     (update-nth
      *mem-tablei*
      (update-nth (ash i -18)
                  (nth *mem-array-next-addr* x86-64)
                  (nth *mem-tablei* x86-64))
      (update-nth
       *mem-array-next-addr*
       (+ *2^18*
          (nth *mem-array-next-addr* x86-64))
       (update-nth
        *mem-arrayi*
        (resize-list (nth *mem-arrayi* x86-64)
                     new-len
                     0)
        x86-64))))))
  :hints (("Goal" :use
           ((:instance
             x86-64p-!memi-new-page-no-resize
             (x86-64 (update-nth *mem-arrayi*
                                 (resize-list (nth *mem-arrayi* x86-64)
                                              new-len 0)
                                 x86-64)))))))

(defthm len-mem-array-positive
  (implies (x86-64p x86-64)
           (posp (len (nth *mem-arrayi* x86-64))))
  :hints (("Goal" :in-theory (enable x86-64p)))
  :rule-classes :type-prescription)

(defthm x86-64p-!memi-new-page
  (implies (and (x86-64p x86-64)
                (n45p i)
                (n64p v)
                (equal (mem-tablei (ash i -18) x86-64)
                       1))
           (mv-let (addr x86-64)
                   (add-page (ash i -18) x86-64)
                   (x86-64p (!mem-arrayi (logior addr
                                                 (logand #x3ffff i))
                                         v
                                         x86-64))))
  :hints (("Goal" :use (x86-64p-!memi-new-page-no-resize
                        x86-64p-!memi-new-page-resize)))
  :rule-classes nil)

(defthm x86-64p-!memi
  (implies (forced-and (x86-64p x86-64)
                       (n45p i) ; quadword address
                       (n64p v))
           (x86-64p (!memi i v x86-64)))
  :hints (("Goal"
           :use (x86-64p-!memi-new-page x86-64p-!memi-not-new-page)
           :in-theory (enable !memi))))

#|| Relics of 32-bit model -- might need to be restored suitably.

; SEG-BASE register update lemmas

(defthm seg-basep-update-nth
  (implies (and (seg-basep x)
                (integerp i)
                (<= 0 i)
                (< i (len x))
                (n64p v))
           (seg-basep (update-nth i v x))))

; SEG-LIMIT register update lemmas

(defthm seg-limitp-update-nth
  (implies (and (seg-limitp x)
                (integerp i)
                (<= 0 i)
                (< i (len x))
                (n20p v))
           (seg-limitp (update-nth i v x))))
```

```
(defthm x86-64p-!seg-limiti
  (implies (and (x86-64p x86-64)
                (integerp i)
                (<= 0 i)
                (< i 6)
                (n20p v))
           (x86-64p (!seg-limiti i v x86-64))))

; SEG-ACCESS register update lemmas

(defthm seg-accessp-update-nth
  (implies (and (seg-accessp x)
                (integerp i)
                (<= 0 i)
                (< i (len x))
                (n12p v))
           (seg-accessp (update-nth i v x))))

(defthm x86-64p-!seg-accessi
  (implies (and (x86-64p x86-64)
                (integerp i)
                (<= 0 i)
                (< i 6)
                (n12p v))
           (x86-64p (!seg-accessi i v x86-64))))

||#

; Should we have this next group of lemmas?  Probably not, but the first, for
; instance, allows later lemma X86-64P-WMB-NO-WRAP (see below) to be proven.

(defthm len-x86-64-rgf
  (implies (x86-64p x86-64)
           (equal (len (nth *rgfi* x86-64))
                  *m86-64-reg-names-len*))
  :hints (("Goal" :in-theory (enable x86-64p x86-64p-pre))))

(defthm len-x86-64-seg
  (implies (x86-64p x86-64)
           (equal (len (nth *segi* x86-64))
                  *m86-64-segment-reg-names-len*))
  :hints (("Goal" :in-theory (enable x86-64p x86-64p-pre))))

(defthm len-x86-64-str
  (implies (x86-64p x86-64)
           (equal (len (nth *stri* x86-64))
                  *m86-64-gdtr-idtr-names-len*))
  :hints (("Goal" :in-theory (enable x86-64p x86-64p-pre))))

(defthm len-x86-64-ssr
  (implies (x86-64p x86-64)
          (equal (len (nth *ssri* x86-64))
                 *m86-64-ldtr-tr-names-len*))
  :hints (("Goal" :in-theory (enable x86-64p x86-64p-pre))))

(defthm len-x86-64-ctr
  (implies (x86-64p x86-64)
           (equal (len (nth *ctri* x86-64))
                  *m86-64-control-reg-names-len*))
  :hints (("Goal" :in-theory (enable x86-64p x86-64p-pre))))

(defthm len-x86-64-dbg
  (implies (x86-64p x86-64)
           (equal (len (nth *dbgi* x86-64))
                  *m86-64-debug-reg-names-len*))
  :hints (("Goal" :in-theory (enable x86-64p x86-64p-pre))))

(defthm len-x86-64-mem
  (implies (x86-64p x86-64)
           (equal (len (nth *mem-tablei* x86-64))
                  *mem-table-size*))
  :hints (("Goal" :in-theory (enable x86-64p x86-64p-pre))))

#|| Relics of 32-bit model -- might need to be restored suitably.

(defthm len-x86-64-seg-base
  (implies (x86-64p x86-64)
           (equal (len (nth *seg-basei* x86-64))
                  *m86-64-segment-reg-names-len*)))

(defthm len-x86-64-seg-limit
  (implies (x86-64p x86-64)
           (equal (len (nth *seg-limiti* x86-64))
                  *m86-64-segment-reg-names-len*)))

(defthm len-x86-64-seg-access
  (implies (x86-64p x86-64)
           (equal (len (nth *seg-accessi* x86-64))
                  *m86-64-segment-reg-names-len*)))

||#

(defthm x86-64p-properties
  (implies (x86-64p-pre x86-64)
           (and (true-listp x86-64)
                (equal (len x86-64) 12)

                (rgfp (nth *rgfi* x86-64))
                (equal (len (nth 0 x86-64))
                       *m86-64-reg-names-len*)

                (ripp (nth *rip* x86-64))
                (flgp (nth *flg* x86-64))

                (segp (nth *segi* x86-64))
                (equal (len (nth *segi* x86-64))
                       *m86-64-segment-reg-names-len*)

                (strp (nth *stri* x86-64))
                (equal (len (nth *stri* x86-64))
                       *m86-64-gdtr-idtr-names-len*)

                (ssrp (nth *ssri* x86-64))
                (equal (len (nth *ssri* x86-64))
                       *m86-64-ldtr-tr-names-len*)

                (ctrp (nth *ctri* x86-64))
                (equal (len (nth *ctri* x86-64))
                       *m86-64-control-reg-names-len*)

                (dbgp (nth *dbgi* x86-64))
                (equal (len (nth *dbgi* x86-64))
                       *m86-64-debug-reg-names-len*)

                (mem-tablep (nth *mem-tablei* x86-64))
                (equal (len (nth *mem-tablei* x86-64))
                       *mem-table-size*)

                (mem-arrayp (nth *mem-arrayi* x86-64))
; Consider a conjunct equating (len (nth *mem-arrayi* x86-64)) with something,
; but be careful about looping if we use (mem-array-length x86-64)).

#|| Relics of 32-bit model -- might need to be restored suitably.

                (seg-basep (nth *seg-basei* x86-64))
                (equal (len (nth *seg-basei* x86-64))
                       *m86-64-segment-reg-names-len*)

                (seg-limitp (nth *seg-limiti* x86-64))
                (equal (len (nth *seg-limiti* x86-64))
                       *m86-64-segment-reg-names-len*)

                (seg-accessp (nth *seg-accessi* x86-64))
                (equal (len (nth *seg-accessi* x86-64))
                       *m86-64-segment-reg-names-len*)
||#
                ))
  :hints (("Goal" :in-theory (enable x86-64p-pre
                                     rgfp ripp flgp segp strp ssrp ctrp dbgp
                                     mem-tablep mem-arrayp
#|| Relics of 32-bit model -- might need to be restored suitably.
                                     seg-basep seg-limitp seg-accessp
||#
```

```
                                        )))
  :rule-classes :forward-chaining)

; Additional lemmas to help with later guard proofs.

; Hopefully, we have proven all the facts we need about the X86
; machine state.

(in-theory (disable x86-64p
                    rgfp        rgfi        !rgfi
                    ripp        rip         !rip
                    flgp        flg         !flg
                    strp        stri        !stri
                    ssrp        ssri        !ssri
                    ctrp        ctri        !ctri
                    dbgp        dbgi        !dbgi
                    mem-tablep  mem-tablei  !mem-tablei
                    mem-arrayp  mem-arrayi  !mem-arrayi
                    mem-array-next-addrp
                    mem-array-next-addr
                    !mem-array-next-addr
#|| Relics of 32-bit model -- might need to be restored suitably.
                    seg-basep   seg-basei   !seg-basei
                    seg-limitp  seg-limiti  !seg-limiti
                    seg-accessp seg-accessi !seg-accessi
||#
                    ))


; Read memory byte and memory double-word functions

(encapsulate
 ()

 (local (include-book "arithmetic-5/top" :dir :system))

 (defthm ash-minus
   (implies (and (syntaxp (quotep k))
                 (< addr n*2^k)
                 (natp addr)
                 (equal n (/ n*2^k (expt 2 k)))
                 (natp n))
            (< (ash addr (- k)) n))
   :rule-classes :linear)

 (defthm ash-monotone
   (implies (and (natp x)
                 (natp y)
                 (natp k))
            (<= (ash (logand x y) k) (ash x k)))
   :rule-classes :linear))
```

```
(encapsulate
 ()

 (local (include-book "arithmetic-5/top" :dir :system))
 (local (include-book "rtl/rel8/lib/top" :dir :system))

 (defthm lemma-1
   (implies (and (integerp x)
                 (<= 0 x)
                 (rationalp y) (<= 0 y)
                 (<= y 1))
            (<= (floor (* x y) 1) x)))

 (defthm ash-negative-is-smaller
   (implies (and (natp x)
                 (integerp k)
                 (<= k 0))
            (<= (ash x k)
                x))
   :hints (("Goal" :in-theory (disable ash-rewrite a2)))
   :rule-classes :linear))

(defthm good-mem-table-entriesp-logic-property-linear
  (let ((addr (nth i mem-table)))
    (implies (and (good-mem-table-entriesp-logic 0 table-max-index
                                                 array-next-addr mem-table)
                  (equal (1- *2^27*) table-max-index)
                  (< addr array-next-addr)
                  (natp i)
                  (<= 0 i)
                  (< i *2^27*)
                  (not (eql addr 1)) ; maybe not needed
                  (<= small #x3ffff))
             (< (+ small addr) array-next-addr)))
  :hints (("Goal"
           :in-theory (disable good-mem-table-entriesp-logic-property)
           :use (:instance good-mem-table-entriesp-logic-property
                           (z 0)
                           (table-max-index (1- *2^27*)))))
  :rule-classes :linear)

(defthm memi-is-unsigned-byte-64
  (implies (and (x86-64p x86-64)
                (n45p addr))
           (n64p (memi addr x86-64)))
  :hints (("Goal" :in-theory (enable memi mem-tablei)))
  :rule-classes
  ((:type-prescription
    :corollary
    (implies (forced-and (x86-64p x86-64)
                        (n45p addr))
            (natp (memi addr x86-64))))
   (:linear
    :corollary
    (implies (forced-and (x86-64p x86-64)
                         (n45p addr))
             (< (memi addr x86-64)
                *2^64*)))))

(defun rm08 (addr x86-64)
  (declare (type (unsigned-byte 48) addr)
           (xargs :guard (x86-64p x86-64)
                  :stobjs (x86-64)))
  (let* ((byte-num     (n03 addr))
         (qword-addr   (ash addr -3))
         (qword        (memi qword-addr x86-64))
         (shift-amount (ash byte-num 3))
         ;; Next form causes (callq (@ .SPBUILTIN-ASH)).
         (shifted-qword (ash qword (- shift-amount))))
    (declare (type (unsigned-byte  3) byte-num)
             (type (unsigned-byte 45) qword-addr)
             (type (unsigned-byte 64) qword shifted-qword)
             (type (integer 0     56) shift-amount))
    (n08 shifted-qword)))

(defthm natp-rm08
  (natp (rm08 addr x86-64))
  :rule-classes :type-prescription)

(defthm rm08-less-than-2^8
  (< (rm08 addr x86-64) 256)
  :rule-classes :linear)

(in-theory (disable rm08))

(defmacro n48+! (x y)
  `(mbe :logic (n48 (+ ,x ,y))
        :exec (+ ,x ,y)))

(defun rm16 (addr x86-64)
  (declare (type (unsigned-byte 48) addr)
           (xargs :guard (and (x86-64p x86-64)
                              (n48p (1+ addr)))
                  :stobjs (x86-64)))
  (let ((byte-num (n03 addr)))
    (declare (type (unsigned-byte 3) byte-num))
    (case byte-num
      (7
       ;; Memory "wrap"
       (let* ((byte0 (rm08        addr    x86-64))
              (byte1 (rm08 (n48+! 1 addr) x86-64)))
         (declare (type (unsigned-byte 8) byte0 byte1))
         (logior (ash byte1 8) byte0)))
      (otherwise
       (let* ((qword-addr   (ash addr -3))
              (qword        (memi qword-addr x86-64))
              (shift-amount (ash byte-num 3))
              (shifted-qword (ash qword (- shift-amount))))
         (declare (type (unsigned-byte 45) qword-addr)
                  (type (unsigned-byte 64) qword shifted-qword)
                  (type (integer 0     56) ; could reduce to 48 with proof work
                        shift-amount))
         (n16 shifted-qword))))))

(defthm natp-rm16
  (natp (rm16 addr x86-64))
  :rule-classes :type-prescription)

(defthm rm16-less-than-2^16
  (< (rm16 addr x86-64) *2^16*)
  :hints (("Goal" :in-theory (enable nxyp-expensive-linears)))
  :rule-classes :linear)

(in-theory (disable rm16))

(defun rm32 (addr x86-64)
  (declare (type (unsigned-byte 48) addr)
           (xargs :guard (and (x86-64p x86-64)
                              (n48p (+ 3 addr)))
                  :stobjs (x86-64)))
  (let ((byte-num (n03 addr)))
    (declare (type (unsigned-byte 3) byte-num))
    (cond
     ((<= byte-num 4)
      (let* ((qword-addr   (ash addr -3))
             (qword        (memi qword-addr x86-64))
             (shift-amount (ash byte-num 3))
             (shifted-qword (ash qword (- shift-amount))))
        (declare (type (unsigned-byte 45) qword-addr)
                 (type (unsigned-byte 64) qword shifted-qword)
                 (type (integer 0     56) ; could reduce with proof work
                       shift-amount))
        (n32 shifted-qword)))
     (t ; byte-num is 5, 6, or 7
      (let* ((word0 (rm16        addr    x86-64))
             (word1 (rm16 (n48+! 2 addr) x86-64)))
        (declare (type (unsigned-byte 16) word0 word1))
        (logior (ash word1 16) word0))))))

(defthm natp-rm32
  (natp (rm32 addr x86-64))
  :rule-classes :type-prescription)
```

```
(defthm rm32-less-than-2^32
  (< (rm32 addr x86-64) *2^32*)
  :hints (("Goal" :in-theory (enable nxyp-expensive-linears)))
  :rule-classes :linear)

(in-theory (disable rm32))

(defun rm64 (addr x86-64)
  (declare (type (unsigned-byte 48) addr)
           (xargs :guard (and (x86-64p x86-64)
                              (n48p (+ 7 addr)))
                  :stobjs (x86-64)))
  (let ((byte-num (n03 addr)))
    (declare (type (unsigned-byte 3) byte-num))
    (cond
     ((eql byte-num 0)
      (let ((qword-addr   (ash addr -3)))
        (declare (type (unsigned-byte 45) qword-addr))
        (memi qword-addr x86-64)))
     (t
      (let* ((dword0 (rm32           addr x86-64))
             (dword1 (rm32 (n48+! 4 addr) x86-64)))
        (declare (type (unsigned-byte 32) dword0 dword1))
        (logior (ash dword1 32) dword0))))))

(in-theory

; At one time (perhaps with slightly different definitions above), disabling
; logior-with-multiple-of-2^18-commuted substantially sped up the proofs of
; natp-rm64 and rm64-less-than-2^64, as follows.

;                        Real time (s)   Run time (s)
; natp-rm64              15.64 -> 0.44   15.05 -> 0.27
; rm64-less-than-2^64    60.45 -> 0.32   58.98 -> 0.26

 (disable logior-with-multiple-of-2^18 logior-with-multiple-of-2^18-commuted))

(defthm natp-rm64
  (implies (and (force (x86-64p x86-64))

; I got a warning about non-recursive NOT when I tried (force (n48p addr)).

                (force (integerp addr))
                (force (<= 0 addr))
                (force (< addr 281474976710656)))
           (natp (rm64 addr x86-64)))
  :rule-classes :type-prescription)

(defthm rm64-less-than-2^64
  (implies (and (force (x86-64p x86-64))
; I got a warning about non-recursive NOT when I tried (force (n48p addr)).

                (force (integerp addr))
                (force (<= 0 addr))
                (force (< addr 281474976710656)))
           (< (rm64 addr x86-64) *2^64*))
  :hints (("Goal" :in-theory (enable nxyp-expensive-linears)))
  :rule-classes :linear)

(in-theory (disable rm64))

(defun rm128 (addr x86-64)
  (declare (type (unsigned-byte 48) addr)
           (xargs :guard (and (x86-64p x86-64)
                              (n48p (+ 15 addr)))
                  :stobjs (x86-64)))
  (let* ((qword0 (rm64           addr x86-64))
         (qword1 (rm64 (n48+! 8 addr) x86-64)))
    (declare (type (unsigned-byte 64) qword0 qword1))
    (logior (ash qword1 64) qword0)))

(defthm natp-rm128
  (implies (and (force (x86-64p x86-64))

; I got a warning about non-recursive NOT when I tried (force (n48p addr)).

                (force (integerp addr))
                (force (<= 0 addr))
                (force (< addr 281474976710656)))
           (natp (rm128 addr x86-64)))
  :rule-classes :type-prescription)

(defthm rm128-less-than-2^128
  (implies (and (force (x86-64p x86-64))

; I got a warning about non-recursive NOT when I tried (force (n48p addr)).

                (force (integerp addr))
                (force (< 0 addr))
                (force (< addr 281474976710656)))
           (< (rm64 addr x86-64) *2^128*))
  :rule-classes :linear)

(in-theory (disable rm128))

; Start write functions.

; At one time (perhaps with slightly different definitions above), the
; following spd up admission of wm08 substantially (12.71 seconds down to 0.64
; seconds).  Disabling just one of these two didn't help nearly as much, though
; (down to roughly 5 or 7 seconds).
(in-theory (disable logior-less-than-2^n logxor-<=-expt-2-to-n))

(encapsulate
 ()

 (local
  (encapsulate
   ()

   (local (include-book "arithmetic-5/top" :dir :system))

   (defthm wm08-lemma-1
     (implies (and (natp addr)
                   (n08p byte))
              (<= (* byte (expt 2 (* 8 (logand 7 addr))))
                  (* byte (expt 2 56))))
     :rule-classes (:rewrite :linear))))

 (local
  (defthm wm08-lemma-2
    (implies (and (natp addr)
                  (n08p byte))
             (< (* 255 (expt 2 (* 8 (logand 7 addr))))
                *2^64*))
    :hints (("Goal"
             :in-theory (disable wm08-lemma-1)
             :use ((:instance wm08-lemma-1 (byte 255)))))
    :rule-classes :linear))

 (defun wm08 (addr byte x86-64)
   (declare (type (unsigned-byte 48) addr)
            (type (unsigned-byte  8) byte)
            (xargs :guard (x86-64p x86-64)
                   :guard-hints
                   (("Goal" :in-theory (enable nxyp-expensive-linears)))
                   :stobjs (x86-64)))
   (let* ((byte-num       (n03 addr))
          (qword-addr     (ash addr -3))
          (qword          (memi qword-addr x86-64))
          (shift-amount   (ash byte-num 3))
          (byte-mask      (ash #xff shift-amount))
          (qword-masked   (logand (lognot byte-mask) qword))
          (byte-to-write  (ash byte shift-amount))
          (qword-to-write (logior qword-masked byte-to-write)))
     (declare (type (unsigned-byte  3) byte-num)
              (type (unsigned-byte 45) qword-addr)
              (type (integer 0     56) shift-amount)
              (type (unsigned-byte 64)
                    qword byte-mask qword-masked byte-to-write qword-to-write))
     (!memi qword-addr qword-to-write x86-64)))
 (defthm x86-64p-wm08
   (implies (forced-and (x86-64p x86-64)
                        (n48p addr)
                        (n08p byte))
            (x86-64p (wm08 addr byte x86-64)))
   :hints (("Goal" :in-theory (enable n64p-logior-n64p-less-than-2^64)))))

(in-theory (disable wm08))

(defthm ash-minus-for-logand
  (implies (and (syntaxp (quotep k))
                (< addr n*2^k)
                (natp addr)
                (natp x)
                (equal n (/ n*2^k (expt 2 k)))
                (natp n))
           (< (ash (logand x addr) (- k)) n))
  :hints (("Goal"
           :use ((:instance ash-minus (addr (logand x addr))))
           :in-theory (disable ash-minus)))
  :rule-classes :linear)

(encapsulate
 ()

 (local
  (encapsulate
   ()

   (local (include-book "arithmetic-5/top" :dir :system))

   (defthm wm16-lemma-1
     (implies (and (natp addr)
                   (not (equal (logand 7 addr) 7))
                   (natp word))
              (<= (* word (expt 2 (* 8 (logand 7 addr))))
                  (* word (expt 2 48))))
     :rule-classes (:rewrite :linear))))

 (local
  (defthm wm16-lemma-2
    (implies (and (natp addr)
                  (not (equal (logand 7 addr) 7)))
             (< (* #xffff (expt 2 (* 8 (logand 7 addr))))
                *2^64*))
    :hints (("Goal"
             :in-theory (disable wm16-lemma-1)
             :use ((:instance wm16-lemma-1 (word #xffff)))))
    :rule-classes :linear))
 (defun wm16 (addr word x86-64)
   (declare (type (unsigned-byte 48) addr)
            (type (unsigned-byte 16) word)
            (xargs :guard (and (x86-64p x86-64)
                               (n48p (1+ addr)))
                   :guard-hints
                   (("Goal" :in-theory (enable nxyp-expensive-linears)))
                   :stobjs (x86-64)))
   (let ((byte-num (n03 addr)))
     (case byte-num
       (7
        ;; Memory "wrap"
        (let* ((x86-64 (wm08 addr (logand word #xff) x86-64))
               (x86-64 (wm08 (n48+! 1 addr)
                             (ash (logand word #xff00) -8)
                             x86-64)))
          x86-64))
       (otherwise
        (let* ((qword-addr     (ash addr -3))
               (qword          (memi qword-addr x86-64))
               (shift-amount   (ash byte-num 3))
               (word-mask      (ash #xffff shift-amount))
               (qword-masked   (logand (lognot word-mask) qword))
               (word-to-write  (ash word shift-amount))
               (qword-to-write (logior qword-masked word-to-write)))
          (declare (type (unsigned-byte 45) qword-addr)
                   (type (integer 0     56) shift-amount)
                   (type (unsigned-byte 64)
                         qword word-mask qword-masked word-to-write
                         qword-to-write))
          (!memi qword-addr qword-to-write x86-64))))))

 (defthm x86-64p-wm16
   (implies (forced-and (x86-64p x86-64)
                        (n48p addr)
                        (n48p (+ 1 addr))
                        (n16p word))
            (x86-64p (wm16 addr word x86-64)))
   :hints (("Goal" :in-theory (enable n64p-logior-n64p-less-than-2^64)))))

(in-theory (disable wm16))

(encapsulate
 ()

 (local
  (encapsulate
   ()

   (local (include-book "arithmetic-5/top" :dir :system))
   (defthm wm32-lemma-1
     (implies (and (natp addr)
                   (<= (logand 7 addr) 4)
                   (natp dword))
              (<= (* dword (expt 2 (* 8 (logand 7 addr))))
                  (* dword (expt 2 32))))
     :rule-classes (:rewrite :linear))

   (defthm wm32-lemma-2
     (implies (and (natp addr)
                   (<= (logand 7 addr) 4))
              (< (* #xffffffff (expt 2 (* 8 (logand 7 addr))))
                 *2^64*))
     :hints (("Goal"
              :in-theory (disable wm32-lemma-1)
              :use ((:instance wm32-lemma-1 (dword #xffffffff)))))
     :rule-classes :linear)))

 (defun wm32 (addr dword x86-64)
   (declare (type (unsigned-byte 48) addr)
            (type (unsigned-byte 32) dword)
            (xargs :guard (and (x86-64p x86-64)
                               (n48p (+ 3 addr)))
                   :guard-hints
                   (("Goal" :in-theory (enable nxyp-expensive-linears)))
                   :stobjs (x86-64)))
   (let ((byte-num (n03 addr)))
     (cond
      ((<= byte-num 4)
       (let* ((qword-addr     (ash addr -3))
              (qword          (memi qword-addr x86-64))
              (shift-amount   (ash byte-num 3))
              (dword-mask     (ash #xffffffff shift-amount))
              (qword-masked   (logand (lognot dword-mask) qword))
              (dword-to-write (ash dword shift-amount))
              (qword-to-write (logior qword-masked dword-to-write)))
         (declare (type (unsigned-byte 45) qword-addr)
                  (type (integer 0     56) shift-amount)
                  (type (unsigned-byte 64)
                        qword dword-mask qword-masked dword-to-write
                        qword-to-write))
         (!memi qword-addr qword-to-write x86-64)))
      (t ; byte-num is 5, 6, or 7
       (let* ((x86-64 (wm16 addr (logand dword #xffff) x86-64))
              (x86-64 (wm16 (n48+! 2 addr)
                            (ash (logand dword #xffff0000) -16)
                            x86-64)))
         x86-64)))))

 (defthm x86-64p-wm32
   (implies (forced-and (x86-64p x86-64)
                        (n48p addr)
                        (n48p (+ 3 addr))
                        (n32p dword))
            (x86-64p (wm32 addr dword x86-64)))
   :hints (("Goal" :in-theory (enable n64p-logior-n64p-less-than-2^64)))))

(in-theory (disable wm32))

(defun wm64 (addr qword x86-64)
  (declare (type (unsigned-byte 48) addr)
           (type (unsigned-byte 64) qword)
           (xargs :guard (and (x86-64p x86-64)
                              (n48p (+ 7 addr)))
                  :stobjs (x86-64)))
  (let ((byte-num (n03 addr)))
    (cond
     ((eql byte-num 0)
      (let ((qword-addr (ash addr -3)))
        (!memi qword-addr qword x86-64)))
     (t
      (let* ((x86-64 (wm32 addr (logand qword #xffffffff) x86-64))
             (x86-64 (wm32 (n48+! 4 addr)
                           (ash (logand qword #xffffffff00000000) -32)
                           x86-64)))
        x86-64)))))

(defthm x86-64p-wm64
  (implies (forced-and (x86-64p x86-64)
                       (n48p addr)
                       (n48p (+ 7 addr))
                       (n64p qword))
           (x86-64p (wm64 addr qword x86-64)))
  :hints (("Goal" :in-theory (enable n64p-logior-n64p-less-than-2^64))))

(in-theory (disable wm64))

; A test

(defun initialize-mem-table (i x86-64)
  (declare (xargs :stobjs x86-64
                  :guard (and (natp i)
                              (<= i *mem-table-size*))
                  :measure (nfix (- *mem-table-size* i))))
  (cond ((mbe :logic (or (not (natp i))
                         (>= i *mem-table-size*))
              :exec (eql i *mem-table-size*))
         x86-64)
        (t (let ((x86-64 (!mem-tablei i 1 x86-64)))
             (initialize-mem-table (1+ i) x86-64)))))

(defun initialize-mem-array (i x86-64)
  (declare (xargs :stobjs x86-64
                  :guard (and (natp i)
                              (<= i (mem-array-length x86-64)))
                  :hints (("Goal" :in-theory (enable !mem-arrayi)))
                  :guard-hints (("Goal" :in-theory (enable !mem-arrayi)))
                  :measure (nfix (- (mem-array-length x86-64) i))))
  (cond ((mbe :logic (or (not (natp i))
                         (>= i (mem-array-length x86-64)))
              :exec (eql i (mem-array-length x86-64)))
         x86-64)
        (t (let ((x86-64 (!mem-arrayi i *default-mem-value* x86-64)))
             (initialize-mem-array (1+ i) x86-64)))))

(defun initialize-x86-64 (x86-64)
  (declare (xargs :stobjs x86-64))
  (let* ((x86-64 (initialize-mem-table 0 x86-64))
         (x86-64 (initialize-mem-array 0 x86-64))
         (x86-64 (!mem-array-next-addr 0 x86-64)))
    x86-64))

(defthm x86-64p-pre-!mem-tablei
  (implies (and (x86-64p-pre x86-64)
                (natp i)
                (< i *mem-table-size*)
                (n45p v))
           (x86-64p-pre (!mem-tablei i v x86-64)))
  :hints (("Goal" :in-theory (enable !mem-tablei x86-64p-pre))))

(defthm x86-64p-pre-initialize-mem-table
  (implies (x86-64p-pre x86-64)
           (x86-64p-pre (initialize-mem-table i x86-64))))

(defthm x86-64p-pre-!mem-arrayi
  (implies (and (x86-64p-pre x86-64)
                (natp i)
                (< i (mem-array-length x86-64))
                (n64p v))
           (x86-64p-pre (!mem-arrayi i v x86-64)))
  :hints (("Goal" :in-theory (enable !mem-arrayi x86-64p-pre))))

(defthm x86-64p-pre-initialize-mem-array
  (implies (x86-64p-pre x86-64)
           (x86-64p-pre (initialize-mem-array i x86-64))))

(defthm x86-64p-pre-initialize-mem-array-next-addr
  (implies (x86-64p-pre x86-64)
           (x86-64p-pre (update-nth *mem-array-next-addr* 0 x86-64)))
  :hints (("Goal" :in-theory (enable x86-64p-pre))))

(defthm mem-array-length-initialize-mem-array
  (equal (len (nth *mem-arrayi*
                  (initialize-mem-array i x86-64)))
         (len (nth *mem-arrayi* x86-64)))
  :hints (("Goal" :in-theory (enable !mem-arrayi))))

(defthm mem-array-length-initialize-mem-table
  (equal (len (nth *mem-arrayi*
                  (initialize-mem-table i x86-64)))
         (len (nth *mem-arrayi* x86-64)))
  :hints (("Goal" :in-theory (enable !mem-tablei))))

(defthm expected-mem-array-next-addr-depends-only-on-mem-table
  (implies (equal (nth *mem-tablei* x86-64-alt)
                  (nth *mem-tablei* x86-64))
           (equal (expected-mem-array-next-addr i j x86-64-alt)
                  (expected-mem-array-next-addr i j x86-64)))
  :hints (("Goal" :in-theory (enable expected-mem-array-next-addr
                                     mem-tablei)))
  :rule-classes nil)

(defthm expected-mem-array-next-addr-!mem-array-next-addr
  (equal (expected-mem-array-next-addr
          i j
          (update-nth *mem-array-next-addr* k x86-64))
         (expected-mem-array-next-addr i j x86-64))
  :hints (("Goal"
           :use
           ((:instance
             expected-mem-array-next-addr-depends-only-on-mem-table
             (x86-64-alt (update-nth *mem-array-next-addr* k x86-64)))))))

(defthm mem-table-initial-mem-array
  (equal (nth *mem-tablei*
              (initialize-mem-array k x86-64))
         (nth *mem-tablei* x86-64))
  :hints (("Goal" :in-theory (enable !mem-arrayi))))

(defthm expected-mem-array-next-addr-initialize-mem-array
  (equal (expected-mem-array-next-addr
          i j
          (initialize-mem-array k x86-64))
         (expected-mem-array-next-addr i j x86-64))
  :hints (("Goal"
           :use
           ((:instance
             expected-mem-array-next-addr-depends-only-on-mem-table
             (x86-64-alt (initialize-mem-array k x86-64)))))))

(defthm mem-tablei-initialize-mem-table
  (implies (and (natp i)
                (natp j)
                (< i *2^27*))
           (equal (nth i (nth *mem-tablei* (initialize-mem-table j x86-64)))
                  (if (<= j i)
                      1
                    (nth i (nth *mem-tablei* x86-64)))))
  :hints (("Goal" :in-theory (enable !mem-tablei mem-tablei))))

(defthm expected-mem-array-next-addr-initialize-mem-table
  (equal (expected-mem-array-next-addr
          i *mem-table-size*
          (initialize-mem-table i x86-64))
         0)
  :hints (("Goal" :in-theory (enable expected-mem-array-next-addr
                                     mem-tablei !mem-tablei))))

(defthm good-mem-table-entriesp-logic-initialize-mem-table
  (implies (and (natp i)
                (<= i bound)
                (natp bound)
                (< bound *mem-table-size*))
           (good-mem-table-entriesp-logic i bound array-next-addr
                                          (nth *mem-tablei*
                                               (initialize-mem-table i
                                                                     X86-64))))
  :hints (("Goal"
           :in-theory (enable good-mem-table-entriesp-logic !mem-tablei))))

; Start proof of good-mem-table-no-dupsp-logic-initialize-mem-table

(defthm empty-initial-mem-table-entries-logic
  (implies (and (natp lower)
                (natp upper)
                (<= lower upper)
                (< upper *mem-table-size*))
           (equal (mem-table-entries-logic
                   lower upper
                   (nth *mem-tablei*
                        (initialize-mem-table 0 x86-64))
                   parity)
                  nil))
  :hints (("Goal" :in-theory (enable mem-table-entries-logic
                                     mem-tablei !mem-tablei))))

(defthm good-mem-table-no-dupsp-logic-initialize-mem-table
  (implies (and (natp i)
                (< i *mem-table-size*)
                (equal bound (1- *mem-table-size*)))
           (good-mem-table-no-dupsp-logic
            i bound
            (nth *mem-tablei*
                 (initialize-mem-table 0 x86-64))))
  :hints (("Goal" :in-theory (enable good-mem-table-no-dupsp-logic))))
```

```
(defun x86-64p-weak (x86-64) ; guard for initialize-x86-64
  (declare (xargs :stobjs x86-64))
  (and (x86-64p-pre x86-64)
       (let ((len (mem-array-length x86-64)))
         (and (equal (logand #x3ffff len) 0)
              (<= *initial-mem-array-length* len)))))

(defthm good-mem-arrayp-1-logic-initialize-mem-array-lemma
  (implies (and (natp i)
                (natp j)
                (< (max i j) (len (nth *mem-arrayi* x86-64))))
           (equal (nth j (nth *mem-arrayi*
                              (initialize-mem-array i x86-64)))
                  (if (<= i j)
                      0
                    (nth j (nth *mem-arrayi* x86-64)))))
  :hints (("Goal" :in-theory (enable !mem-arrayi))))

(defthm good-mem-arrayp-1-logic-initialize-mem-array
  (implies (and (natp i)
                (natp len)
                (<= i len)
                (equal mem-array (nth *mem-arrayi* x86-64))
                (<= len (len mem-array)))
           (good-mem-arrayp-1-logic
            i len
            (nth *mem-arrayi*
                 (initialize-mem-array 0 x86-64)))))

(defthm x86-64p-initialize-x86-64
  (implies (x86-64p-weak x86-64)
           (x86-64p (initialize-x86-64 x86-64)))
  :hints (("Goal" :in-theory (enable x86-64p
                                     mem-array-next-addr
                                     !mem-array-next-addr))))

(in-theory (disable initialize-x86-64 x86-64p-weak))

(defun test-x86-64 (x86-64 init-p)
  (declare (xargs :stobjs x86-64
                  :guard (if init-p
                             (x86-64p-weak x86-64)
                           (x86-64p x86-64))))
  (let* ((x86-64 (if init-p
                     (initialize-x86-64 x86-64)
                   x86-64))
         (val-0 (rm64 3 x86-64))
         (x86-64 (wm64 0 0 x86-64))
         (x86-64 (wm64 8 0 x86-64))
         (x86-64 (wm64 16 0 x86-64)) ; maybe not necessary
         (x86-64                      ; 01
          (wm08 0 #x01 x86-64))
         (val-0100 (rm16 0 x86-64))
         (x86-64 ; 014523
          (wm16 1 #x2345 x86-64))
         (val-45 (rm08 1 x86-64))
         (val-4501 (rm16 0 x86-64))
         (x86-64 ; 01458967
          (wm16 2 #x6789 x86-64))
         (val-67894501 (rm32 0 x86-64))
         (x86-64 ; 01458967_1100ffee_ddccbbaa
          (wm64 4 #xaabbccddeeff0011 x86-64))
         (val-1167 (rm16 3 x86-64))
         (x86-64 ; 01238967_1100ffee_88776655_44332211
          (wm64 8 #x1122334455667788 x86-64))
         (val-1122334455667788
          (rm64 8 x86-64))
         (val-55667788eeff0011
          (rm64 4 x86-64))
         (x86-64 ; 01238967_ddccbbaa_88776655_44332211
          (wm32 4 #xaabbccdd x86-64))
         (val-aabbccdd
          (rm32 4 x86-64))
         (page-12-addr (* 12 *2^21*)) ; page boundary
         (x86-64
          (wm64 (- page-12-addr 3) #x1122334455667788 x86-64))
         (val-last-all ; #x1122334455667788
          (rm64 (- page-12-addr 3) x86-64))
         (val-last-lower ; #x55667788
          (rm32 (- page-12-addr 3) x86-64))
         (val-last-upper ; #x11223344
          (rm32 (+ page-12-addr 1) x86-64)))
    (mv (and (equal val-0 0)
             (equal val-0100 #x0001)
             (equal val-45 #x45)
             (equal val-4501 #x4501)
             (equal val-67894501 #x67894501)
             (equal val-1167 #x1167)
             (equal val-1122334455667788 #x1122334455667788)
             (equal val-55667788eeff0011 #x55667788eeff0011)
             (equal val-aabbccdd #xaabbccdd)
             (equal val-last-all #x1122334455667788)
             (equal val-last-lower #x55667788)
             (equal val-last-upper #x11223344))
        x86-64)))

(defun my-test (init-p)
  (with-local-stobj x86-64
                    (mv-let (ans x86-64)
                            (test-x86-64 x86-64 init-p)
                            ans)))
```

```
(assert-event (my-test t))
```

12 ========== File x86-64/read-over-write-proofs.lisp =========

```
;;; !!! TO DO:

;;; Prove analogues of rgfi-!rgfi for other stobj fields (should be easy).

;;; Consider proving and exporting "-same" and "-different" lemmas, such as the
;;; following.

#||
; Exported theorem:
(defthm rgfi-!rgfi-same
  (equal (rgfi i (!rgfi i v x86-64))
         v)
  :hints (("Goal" :in-theory (enable rgfi !rgfi))))

; Exported theorem:
(defthm rgfi-!rgfi-different
  (implies (and (force (n03p i))
                (force (n03p j))
                (not (equal i j)))
           (equal (rgfi i (!rgfi j v x86-64))
                  (rgfi i x86-64)))
  :hints (("Goal" :in-theory (enable rgfi !rgfi))))
||#

(in-package "ACL2")

; All theorems in this file are local except for those that are to be exported,
; which are labeled with the comment "; Exported theorem:".  We use def-gl-thm
; (actually a local version) to dispatch lemmas that were seen to be needed
; upon analysis of simplification checkpoints.  Without def-gl-thm, we would
; consider using one of these libraries:

; (local (include-book "arithmetic-5/top" :dir :system))
; (local (include-book "rtl/rel8/lib/top" :dir :system))

; That would perhaps take considerable work, however, because wm08 uses lognot,
; which doesn't appear in many lemmas in the above libraries.  Less important
; is that we might have to disable some rules, e.g. ash-rewrite in the latter
; case.  Fortunately, finding lemmas in either of the above books isn't hard,
; for example using the find-lemmas utility:

; (local (include-book "misc/find-lemmas" :dir :system))

(include-book "x86-state")

(local (include-book "centaur/gl/gl" :dir :system))

(defmacro def-local-gl-thm (&rest args)
  `(local (def-gl-thm ,@args)))
```

```
; Exported theorem:
(defthm rgfi-!rgfi
  (implies (and (force (n03p i))
                (force (n03p j))
                (not (equal i j)))
           (equal (rgfi i (!rgfi j v x86-64))
                  (if (equal i j)
                      v
                    (rgfi i x86-64))))
  :hints (("Goal" :in-theory (enable rgfi !rgfi))))

(in-theory (disable (resize-list))) ; prevent stack overflow

(local
 (defthm memi-!memi-same
   (implies (force (x86-64p x86-64))
            (equal (memi i (!memi i v x86-64))
                   v))
   :hints (("Goal" :in-theory (enable memi !memi
                                       mem-tablei !mem-tablei
                                       mem-arrayi !mem-arrayi
                                       mem-array-next-addr)))))

; Start proof of memi-!memi-different

(def-local-gl-thm equal-logior-with-page-aligned
  :hyp (and (n18p x1)
            (n18p x2)
            (n45p y) ; (natp y) should suffice
            (equal (logand #x3ffff y) 0)
            (n45p z) ; (natp z) should suffice
            (equal (logand #x3ffff z) 0))
  :concl (equal (equal (logior x1 y)
                       (logior x2 z))
                (and (equal x1 x2)
                     (equal y z)))
  :g-bindings
  `((y (:g-number ,(gl-int  0  2  46)))
    (z (:g-number ,(gl-int  1  2  46)))
    (x1 (:g-number ,(gl-int  92 2 19)))
    (x2 (:g-number ,(gl-int  93 2 19)))))

(def-local-gl-thm equal-logior-with-page-aligned-commuted
  :hyp (and (n18p x1)
            (n18p x2)
            (n45p y) ; (natp y) should suffice
            (equal (logand #x3ffff y) 0)
            (n45p z) ; (natp z) should suffice
            (equal (logand #x3ffff z) 0))
  :concl (equal (equal (logior y x1)
                       (logior z x2))
                (and (equal x1 x2)
                     (equal y z)))
  :g-bindings
  `((y (:g-number ,(gl-int  0  2  46)))
    (z (:g-number ,(gl-int  1  2  46)))
    (x1 (:g-number ,(gl-int  92 2 19)))
    (x2 (:g-number ,(gl-int  93 2 19)))))

(def-local-gl-thm equal-logior-with-page-aligned-commuted-2
  :hyp (and (n18p x1)
            (n18p x2)
            (n45p y) ; (natp y) should suffice
            (equal (logand #x3ffff y) 0)
            (n45p z) ; (natp z) should suffice
            (equal (logand #x3ffff z) 0))
  :concl (equal (equal (logior x1 y)
                       (logior z x2))
                (and (equal x1 x2)
                     (equal y z)))
  :g-bindings
  `((y (:g-number ,(gl-int  0  2  46)))
    (z (:g-number ,(gl-int  1  2  46)))
    (x1 (:g-number ,(gl-int  92 2 19)))
    (x2 (:g-number ,(gl-int  93 2 19)))))

; Start proof of mem-table-is-one-to-one

; The following was somehow disabled when including the gl book (email sent to
; Sol 11/23/11), but it seems reasonable to leave it enabled, for example, to
; prove the next theorem, arith-hack.

(in-theory (enable default-+-2))

(local
 (defthm arith-hack
   (equal (+ 1 -1 i)
          (fix i))))

(local
 (defthm member-mem-table-entries-logic
   (implies (and (natp i)
                 (natp lower)
                 (natp upper)
                 (<= lower i)
                 (<= i upper)
                 (not (equal (nth i mem-table) 1)))
            (member (nth i mem-table)
                    (mem-table-entries-logic lower upper mem-table parity)))
   :hints (("Subgoal *1/7.1" :cases ((equal lower i)))
           ("Subgoal *1/6.1" :cases ((equal lower i)))
           ("Subgoal *1/5.2" :cases ((equal lower i)))
           ("Subgoal *1/4.2" :cases ((equal lower i)))
           ("Subgoal *1/3''" :cases ((equal lower i))))))

(local
 (defun sortedp (x parity)

; A parity of true means that x should be increasing.

   (cond ((or (endp x) (endp (cdr x)))
          t)
         ((if parity
              (<= (car x) (cadr x))
            (>= (car x) (cadr x)))
          (sortedp (cdr x) parity))
         (t nil))))

(local
 (defthm sortedp-revappend
   (iff (and (sortedp x (not parity))
             (sortedp y parity)
             (or (atom x)
                 (atom y)
                 (if parity
                     (<= (car x) (car y))
                   (>= (car x) (car y)))))
        (sortedp (revappend x y) parity))
   :rule-classes ((:rewrite
                   :corollary
                   (equal (sortedp (revappend x y) parity)
                          (and (sortedp x (not parity))
                               (sortedp y parity)
                               (or (atom x)
                                   (atom y)
                                   (if parity
                                       (<= (car x) (car y))
                                     (>= (car x) (car y))))))))))

(local
 (defthm sortedp-merge-<-into->
   (implies (and (sortedp x t)
                 (sortedp y t)
                 (sortedp z nil)
                 (or (atom z) (atom x) (>= (car x) (car z)))
                 (or (atom z) (atom y) (>= (car y) (car z))))
            (sortedp (merge-<-into-> x y z) nil))))

(local
 (defthm sortedp-merge->-into-<
   (implies (and (sortedp x nil)
                 (sortedp y nil)
```

```
                (sortedp z t)
                (or (atom z) (atom x) (<= (car x) (car z)))
                (or (atom z) (atom y) (<= (car y) (car z))))
           (sortedp (merge->-into-< x y z) t))))

(local
 (defthm no-duplicatesp-sorted-revappend
   (equal (no-duplicatesp-sorted (revappend x y))
          (and (no-duplicatesp-sorted x)
               (no-duplicatesp-sorted y)
               (or (atom x)
                   (atom y)
                   (not (equal (car x) (car y))))))))

(local
 (defthm not-member-sortedp-t
   (implies (and (sortedp x t)
                 (< a (car x)))
            (not (member a x)))))

(local
 (defthm member-sortedp-t
   (implies (and (sortedp x t)
                 (consp x)
                 (<= a (car x))
                 (rational-listp x))
            (iff (member a x)
                 (equal a (car x))))
   :hints (("Goal" :induct t))))

(local
 (defthm member-of-both-implies-not-no-duplicatesp-sorted-merge-<-into->
   (implies (and (rational-listp x)
                 (rational-listp y)
                 (member a x)
                 (member b y)
                 (equal a b)
                 (sortedp x t)
                 (sortedp y t))
            (not (no-duplicatesp-sorted (merge-<-into-> x y z))))
   :hints (("Goal"
            :induct (merge-<-into-> x y z)
            :expand ((merge-<-into-> x y z)
                     (sortedp x t)
                     (merge-<-into-> x
                                     (cdr y)
                                     (cons (car x) z)))))))

(local
 (defthm not-member-sortedp-nil
   (implies (and (sortedp x nil)
                 (> a (car x)))
            (not (member a x)))))

(local
 (defthm member-sortedp-nil
   (implies (and (sortedp x nil)
                 (consp x)
                 (>= a (car x))
                 (rational-listp x))
            (iff (member a x)
                 (equal a (car x))))
   :hints (("Goal" :induct t))))

(local
 (defthm member-of-both-implies-not-no-duplicatesp-sorted-merge->-into-<
   (implies (and (rational-listp x)
                 (rational-listp y)
                 (member a x)
                 (member b y)
                 (equal a b)
                 (sortedp x nil)
                 (sortedp y nil))
            (not (no-duplicatesp-sorted (merge->-into-< x y z))))
   :hints (("Goal"
            :induct (merge->-into-< x y z)
            :expand ((merge->-into-< x y z)
                     (sortedp x nil)
                     (merge->-into-< x
                                     (cdr y)
                                     (cons (car x) z)))))))

(local
 (defthm rationalp-nth
   (implies (and (rational-listp x)
                 (natp n)
                 (< n (len x)))
            (rationalp (nth n x)))
   :hints (("Goal" :in-theory (enable nth)))))

(local
 (defthm rational-listp-mem-table-entries-logic
   (implies (and (rational-listp mem-table)
                 (natp lower)
                 (<= lower upper)
                 (natp upper)
                 (< upper (len mem-table)))
            (rational-listp (mem-table-entries-logic lower upper mem-table
                                                     parity)))))

(local
 (defthm sortedp-mem-table-entries-logic
   (implies (and (rational-listp mem-table)
                 (natp lower)
                 (natp upper)
                 (< upper (len mem-table))
                 (booleanp parity))
            (sortedp (mem-table-entries-logic lower upper mem-table parity)
                     parity))))

(local
 (defthm member-mem-table-entries-logic-alt
   (implies (and (equal (nth i mem-table) (nth j mem-table))
                 (natp i)
                 (natp lower)
                 (natp upper)
                 (<= lower i)
                 (<= i upper)
                 (not (equal (nth i mem-table) 1)))
            (member (nth j mem-table)
                    (mem-table-entries-logic lower upper mem-table parity)))))

(local
 (defthm mem-table-is-one-to-one-lemma
   (implies (and (rational-listp mem-table)
                 (natp lower)
                 (natp upper)
                 (natp i)
                 (natp j)
                 (<= lower i)
                 (< i j)
                 (<= j upper)
                 (< upper (len mem-table))
                 (booleanp parity)
                 (not (equal (nth i mem-table) 1))
                 (equal (nth i mem-table)
                        (nth j mem-table)))
            (not (no-duplicatesp-sorted
                  (mem-table-entries-logic lower upper mem-table parity))))
   :hints
   (("Goal"
     :restrict
     ((member-of-both-implies-not-no-duplicatesp-sorted-merge->-into-<
       ((a (nth i mem-table))
        (b (nth j mem-table))))
      (member-of-both-implies-not-no-duplicatesp-sorted-merge-<-into->
       ((a (nth i mem-table))
        (b (nth j mem-table)))))))
   :rule-classes nil))

(local
 (defthm mem-table-is-one-to-one
   (implies (and (and (x86-64p x86-64)
                      (n27p i)
                      (n27p j))
                 (not (equal (nth i (nth *mem-tablei* x86-64))
                             1)))
            (equal (equal (nth i (nth *mem-tablei* x86-64))
                          (nth j (nth *mem-tablei* x86-64)))
                   (equal i j)))
   :hints (("Goal"
            :in-theory (enable x86-64p good-mem-table-no-dupsp-logic)
            :use ((:instance mem-table-is-one-to-one-lemma
                             (lower 0)
                             (upper (1- *2^27*))
                             (mem-table (nth *mem-tablei* x86-64))
                             (parity t))
                  (:instance mem-table-is-one-to-one-lemma
                             (lower 0)
                             (upper (1- *2^27*))
                             (i j) (j i)
                             (mem-table (nth *mem-tablei* x86-64))
                             (parity t)))))))

(def-local-gl-thm equality-as-18-bit-logand-and-ash
  :hyp (and (n45p x)
            (n45p y)
            (not (equal x y))
            (equal (logand #x3ffff x)
                   (logand #x3ffff y)))
  :concl (not (equal (ash x -18) (ash y -18)))
  :g-bindings
  `((x (:g-number ,(gl-int  0  2  46)))
    (y (:g-number ,(gl-int  1  2  46)))))

(local
 (defthm mem-array-next-addr-page-is-0-lemma
   (implies (and (good-mem-arrayp-1-logic mem-array-next-addr
                                          (len mem-array)
                                          mem-array)
                 (natp addr)
                 (natp mem-array-next-addr)
                 (<= mem-array-next-addr addr)
                 (< addr (len mem-array)))
            (equal (nth addr mem-array)
                   0))
   :rule-classes nil))

(local
 (defthm mem-array-next-addr-page-is-0
   (implies (and (x86-64p x86-64)
                 (natp addr)
                 (< addr (len (nth *mem-arrayi* x86-64)))
                 (<= (nth *mem-array-next-addr* x86-64) addr))
```

```
           (equal (nth addr
                       (nth *mem-arrayi* x86-64))
                  0))
  :hints (("Goal"
           :in-theory (enable x86-64p mem-array-next-addr)
           :use ((:instance mem-array-next-addr-page-is-0-lemma
                            (mem-array-next-addr (nth *mem-array-next-addr*
                                                      x86-64))
                            (addr addr)
                            (mem-array (nth *mem-arrayi* x86-64))))))))

(local
 (encapsulate
  ()

  (local (include-book "arithmetic-5/top" :dir :system))

  (defthm logior->=
    (implies (and (natp x) (natp y))
             (and (<= x (logior x y))
                  (<= y (logior x y))))
    :rule-classes :linear)))

(local
 (defthm mem-array-length-is-page-aligned
   (implies (x86-64p x86-64)
            (equal (logand #x3ffff (len (nth *mem-arrayi* x86-64)))
                   0))
   :hints (("Goal" :in-theory (enable x86-64p)))))

(local
 (defthm logior-logand-2^18-commuted
   (implies (and (natp x)
                 (natp y)
                 (natp z)
                 (< x z)
                 (equal (logand #x3ffff x) 0)
                 (equal (logand #x3ffff z) 0))
            (< (logior (logand #x3ffff y)
                       x)
               z))))

(local
 (defthm memi-!memi-different
   (implies (forced-and (not (equal i j))
                        (n45p i)
                        (n45p j)
                        (x86-64p x86-64))
            (equal (memi i (!memi j v x86-64))
                   (memi i x86-64)))
   :hints (("Goal" :in-theory (e/d (memi !memi !mem-array-next-addr
                                         mem-tablei !mem-tablei mem-arrayi
                                         !mem-arrayi mem-array-next-addr)
                                   ((force)))))))

; Exported theorem:
(defthm memi-!memi
  (implies (forced-and (x86-64p x86-64)
                       (n45p i)
                       (n45p j))
           (equal (memi i (!memi j v x86-64))
                  (if (equal i j)
                      v
                    (memi i x86-64)))))

(local (in-theory (disable memi-!memi-same memi-!memi-different)))

(def-local-gl-thm rm08-wm08-same-lemma-1
  :hyp (force (and (n08p v)
                   (n64p mem-val)
                   (n48p i48)))
  :concl (equal
          (logand 255
                  (ash (logior (* v (expt 2 (* 8 (logand 7 i48))))
                               (logand (+ -1
                                          (- (* 255 (expt 2 (* 8 (logand 7 i48))))))
                                       mem-val))
                       (- (* 8 (logand 7 i48)))))
          v)
  :g-bindings
  `((mem-val (:g-number ,(gl-int 0 1 65)))
    (v (:g-number ,(gl-int 65 1 9)))
    (i48 (:g-number ,(gl-int 74 1 49)))))

; From x86-state.lisp:
(def-local-gl-thm ash-addr--2-is-less
    :hyp (n48p i)
    :concl (< (ash i -3) *2^45*)
    :rule-classes :linear
    :g-bindings
    `((i (:g-number ,(gl-int 0 1 65)))))

(local
 (defthm rm08-wm08-same
   (implies (and (force (x86-64p x86-64))
                 (force (n48p i))
                 (force (n08p v)))
            (equal (rm08 i (wm08 i v x86-64))
                   v))
   :hints (("Goal" :in-theory (enable rm08 wm08)))))

(local
 (encapsulate
  ()

  (def-local-gl-thm rm08-wm08-different-same-qword-lemma-1
    :hyp (force (and (n64p mem-val)
                     (n03p i03)
                     (n03p j03)
                     (not (equal i03 j03))
                     (n08p v)))
    :concl (equal
            (logand 255
                    (ash (logior (* v (expt 2 (* 8 j03)))
                                 (logand (+ -1
                                            (- (* 255 (expt 2 (* 8 j03)))))
                                         mem-val))
                         (- (* 8 i03))))
            (logand 255
                    (ash mem-val
                         (- (* 8 i03)))))
    :g-bindings
    `((mem-val
       (:g-number ,(gl-int 0 1 65)))
      (i03
       (:g-number ,(gl-int 65 1 4)))
      (j03
       (:g-number ,(gl-int 69 1 4)))
      (v
       (:g-number ,(gl-int 73 1 9)))))

  (def-local-gl-thm rm08-wm08-different-same-qword-lemma-2
    :hyp (and (n48p i)
              (n48p j)
              (equal (ash j -3) (ash i -3)))
    :concl (equal (EQUAL (LOGAND 7 I) (LOGAND 7 J))
                  (equal j i))
    :g-bindings
    `((i (:g-number ,(gl-int 0 2 49)))
      (j (:g-number ,(gl-int 1 2 49)))))

  (defthm rm08-wm08-different-same-qword
    (implies (and (x86-64p x86-64)
                  (n48p i)
                  (n48p j)
                  (equal (ash i -3) (ash j -3))
                  (not (equal i j))
                  (n08p v))
             (equal (rm08 i (wm08 j v x86-64))
                    (rm08 i x86-64)))
    :hints (("Goal"
             :in-theory (enable rm08 wm08)))
    :rule-classes nil))
```

```
(local
 (defthm rm08-wm08-different-different-qword
   (implies (and (x86-64p x86-64)
                 (n48p i)
                 (n48p j)
                 (not (equal (ash i -3) (ash j -3)))
                 (not (equal i j))
                 (n08p v))
            (equal (rm08 i (wm08 j v x86-64))
                   (rm08 i x86-64)))
   :hints (("Goal" :in-theory (enable rm08 wm08)))
   :rule-classes nil))

(local
 (defthm rm08-wm08-different
   (implies (and (force (x86-64p x86-64))
                 (force (n48p i))
                 (force (n48p j))
                 (not (equal i j))
                 (force (n08p v)))
            (equal (rm08 i (wm08 j v x86-64))
                   (rm08 i x86-64)))
   :hints (("Goal" :use (rm08-wm08-different-same-qword
                         rm08-wm08-different-different-qword)))))

; Exported theorem:
(defthm rm08-wm08
  (implies (and (force (x86-64p x86-64))
                (force (n48p i))
                (force (n48p j))
                (force (n08p v)))
           (equal (rm08 i (wm08 j v x86-64))
                  (if (equal i j)
                      v
                    (rm08 i x86-64)))))

(local (in-theory (disable rm08-wm08-same rm08-wm08-different)))

; Start proof of rm16-wm16-same and rm16-wm16-different.

; Start proof of rm16-as-rm08

; Start proof of rm16-wm16-non-overlap.

(def-local-gl-thm rm16-as-rm08-lemma-1
  :hyp (and (n03p addr03)
            (n64p mem-val))
  :concl (equal (logior (logand 255
                                (ash mem-val
                                     (- (* 8 addr03))))
```

```
                        (* 256
                           (logand 255
                                   (ash mem-val
                                        (- (+ 8 (* 8 addr03)))))))
                (logand 65535
                        (ash mem-val
                             (- (* 8 addr03)))))
  :g-bindings
  `((addr03 (:g-number ,(gl-int 0 1 4)))
    (mem-val (:g-number ,(gl-int 4 1 65)))))

(def-local-gl-thm rm16-as-rm08-lemma-2
  :hyp (and (n64p addr)
            (not (equal (logand 7 addr) 7)))
  :concl (equal (logand 7 (+ 1 addr))
                (+ 1 (logand 7 addr)))
  :g-bindings
  `((addr (:g-number ,(gl-int 0 1 65)))))

(def-local-gl-thm rm16-as-rm08-lemma-3
  :hyp (and (n64p addr)
            (not (equal (logand 7 addr) 7)))
  :concl (equal (ash (+ 1 addr) -3)
                (ash addr -3))
  :g-bindings
  `((addr (:g-number ,(gl-int 0 1 65)))))

; Exported theorem (to be disabled):
(defthm rm16-as-rm08
  (implies (and (force (x86-64p x86-64))
                (force (natp addr))
                (force (n48p (1+ addr))))
           (equal (rm16 addr x86-64)
                  (let ((byte0 (rm08 addr x86-64))
                        (byte1 (rm08 (n48+! 1 addr) x86-64)))
                    (logior (ash byte1 8) byte0))))
  :hints (("Goal" :in-theory (enable rm16 rm08))))

; Start proof of wm16-as-wm08

(local
 (defthm update-nth-update-nth-same
   (equal (update-nth i v1 (update-nth i v2 lst))
          (update-nth i v1 lst))))

(local
 (defthm !memi-!memi-same
   (implies (x86-64p x86-64)
            (equal (!memi addr v1 (!memi addr v2 x86-64))
                   (!memi addr v1 x86-64)))
   :hints (("Goal" :in-theory (enable !memi !mem-arrayi !mem-tablei mem-tablei
                                    mem-array-next-addr)))))

(def-local-gl-thm wm16-as-wm08-lemma-1
  :hyp (and (n03p addr03)
            (n64p mem-val)
            (n16p word))
  :concl
  (equal (logior
          (* (ash (logand 65280 word) -8)
             (expt 2 (+ 8 (* 8 addr03))))
          (logand (+ -1
                     (- (* 255
                           (expt 2 (+ 8 (* 8 addr03))))))
                  (logior (* (logand 255 word)
                             (expt 2 (* 8 addr03)))
                          (logand (+ -1
                                     (- (* 255 (expt 2 (* 8 addr03)))))
                                  mem-val))))
         (logior (* word (expt 2 (* 8 addr03)))
                 (logand (+ -1
                            (- (* 65535 (expt 2 (* 8 addr03)))))
                         mem-val)))
  :g-bindings
  `((addr03 (:g-number ,(gl-int 0 1 4)))
    (mem-val (:g-number ,(gl-int 4 1 65)))
    (word (:g-number ,(gl-int 69 1 17)))))

; Exported theorem (to be disabled):
(defthm wm16-as-wm08
  (implies (forced-and (x86-64p x86-64)
                       (natp addr)
                       (n48p (1+ addr))
                       (n16p word))
           (equal (wm16 addr word x86-64)
                  (let* ((x86-64 (wm08 addr (logand word #xff) x86-64))
                         (x86-64 (wm08 (+ 1 addr)
                                       (ash (logand word #xff00) -8)
                                       x86-64)))
                    x86-64)))
  :hints (("Goal" :in-theory (enable wm16 wm08))))

(def-local-gl-thm rm16-wm16-same-lemma-1
  :hyp (n16p v)
  :concl (equal (logior (logand 255 v)
                        (* 256 (ash (logand 65280 v) -8)))
                v)
  :g-bindings
  `((v (:g-number ,(gl-int 0 1 17)))))

; Exported theorem:
(defthm rm16-wm16
  (implies (and (force (x86-64p x86-64))
                (force (natp i))
                (force (n48p (1+ i)))
                (force (natp j))
                (force (n48p (1+ j)))
                (force (n16p v)))
           (equal (rm16 i (wm16 j v x86-64))
                  (cond ((equal i j)
                         v)
                        ((equal j (1+ i))
                         (logior (* *2^8* (logand #xff v))
                                 (rm08 i x86-64)))
                        ((equal i (1+ j))
                         (logior (ash (logand #xff00 v) -8)
                                 (* *2^8* (rm08 (+ 1 i) x86-64))))
                        (t
                         (rm16 i x86-64))))))
```

13 ========== File x86-64/read-over-write.lisp ==========

```
(in-package "ACL2")

; See read-over-write-proofs.lisp for the proofs.

(include-book "x86-state")
(local (include-book "read-over-write-proofs"))

(set-enforce-redundancy t)

; The following "-as-" lemmas were proved as lemmas towards our main
; read-over-write results for memory.  We leave them disabled, but keep them
; around in case they are of use.

(defthmd rm16-as-rm08
  (implies (and (force (x86-64p x86-64))
                (force (natp addr))
                (force (n48p (1+ addr))))
           (equal (rm16 addr x86-64)
                  (let ((byte0 (rm08 addr x86-64))
                        (byte1 (rm08 (n48+! 1 addr) x86-64)))
                    (logior (ash byte1 8) byte0)))))

(defthmd wm16-as-wm08
  (implies (forced-and (x86-64p x86-64)
                       (natp addr)
                       (n48p (1+ addr))
                       (n16p word))
           (equal (wm16 addr word x86-64)
                  (let* ((x86-64 (wm08 addr (logand word #xff) x86-64))
                         (x86-64 (wm08 (+ 1 addr)
                                       (ash (logand word #xff00) -8)
                                       x86-64)))
                    x86-64)))
  :hints (("Goal" :in-theory (enable wm16 wm08))))

; End of "-as" lemmas.

(defthm rgfi-!rgfi
  (implies (and (force (n03p i))
                (force (n03p j))
                (not (equal i j)))
           (equal (rgfi i (!rgfi j v x86-64))
                  (if (equal i j)
                      v
                    (rgfi i x86-64)))))

(defthm memi-!memi
  (implies (forced-and (x86-64p x86-64)
                       (n45p i)
                       (n45p j))
           (equal (memi i (!memi j v x86-64))
                  (if (equal i j)
                      v
                    (memi i x86-64)))))

(defthm rm08-wm08
  (implies (and (force (x86-64p x86-64))
                (force (n48p i))
                (force (n48p j))
                (force (n08p v)))
           (equal (rm08 i (wm08 j v x86-64))
                  (if (equal i j)
                      v
                    (rm08 i x86-64)))))

(defthm rm16-wm16
  (implies (and (force (x86-64p x86-64))
                (force (natp i))
                (force (n48p (1+ i)))
                (force (natp j))
                (force (n48p (1+ j)))
                (force (n16p v)))
           (equal (rm16 i (wm16 j v x86-64))
                  (cond ((equal i j)
                         v)
                        ((equal j (1+ i))
                         (logior (* *2^8* (logand #xff v))
                                 (rm08 i x86-64)))
                        ((equal i (1+ j))
                         (logior (ash (logand #xff00 v) -8)
                                 (* *2^8* (rm08 (+ 1 i) x86-64))))
                        (t
                         (rm16 i x86-64))))))
```

14 ========== File x86-64/x86-utils.lisp ==========

```
(in-package "ACL2")

(include-book "x86-state")

; Needed for at least a couple of lemmas.  This book certifies quickly, so
; let’s go ahead and enable these.
(local (in-theory (enable nxyp-expensive-linears)))

; Start preliminary utilities.

; We introduce several utilities trafficking in "erp" objects, each of which is
; a stack of individual error objects typically of the form (ctx . kwd-alist),
; where ctx is typically a function name.  We do not enforce the shape of an
; individual error object, however.

(defmacro !ms-erp (&rest args)

; This macro assumes that erp is already bound to an "error" stack, and ctx is
; bound to the current "context", which is typically the instruction (a
; symbol).

  ‘(cons (list ctx ,@args)
         erp))

(defmacro !ms-erp-fresh (&rest args)

; This is same as !ms-erp, but where erp is not bound and we bind it to nil.

  ‘(let ((erp nil))
     (!ms-erp ,@args)))

(defmacro !!ms (&rest args)

; Erp, ctx, and also x86-64 must already be bound.  We return an updated x86-64
; that has a non-nil ms field conveying useful information.  See also !ms-erp.

  ‘(!ms (!ms-erp :rip (rip x86-64) ,@args)
        x86-64))

(defmacro !!ms-fresh (&rest args)

; Ctx must already be bound; see !ms-erp-fresh and !!ms for explanation.

  ‘(!ms (!ms-erp-fresh :rip (rip x86-64) ,@args)
        x86-64))

; Next we present instruction decoding utilities.  Our decoding process
; repeatedly grabs additional bytes from a right-shifted "tail" of the original
; 15 bytes passed to the instruction decoder, x86-64-decode, maintaining a
; count of the number of bytes grabbed, ibytes.

(defmacro itailp (x)
; This macro is true for x a tail of a 15-byte number.  For simplicity we use
; a weak recognizer for this purpose.
  `(natp ,x))

(defmacro ibytesp (x)

; This macro is true for x the number of bytes parsed so far for an
; instruction, which should always be at most 15.  We considered using `(n04p
; ,x) instead.  But then our guards of ibytesp would generate proof obligations
; that force us to track the total number of bytes accumulated, which is extra
; effort that serves no purpose.

  `(natp ,x))

(defmacro rexp (rex)

; We only apply rexp to a rex byte returned by our instruction decoder, which
; is always an 8-bit number: if that decoder encounters #x4-, then we have a
; non-zero rex byte (see x86-64-decode-rex); if it doesn't, then it returns 0
; for the rex byte.

  `(not (zerop ,rex)))

; General purpose register indices are 3 bits except in 64-bit mode, where they
; can have 4 bits depending on the rex prefix.

(defun reg-indexp (reg rex)
  (declare (xargs :guard (n08p rex)))
  (if (rexp rex)
      (n04p reg)
    (n03p reg)))

(defthm reg-indexp-for-3-bits
; We need this for (defstep leave ...), for (reg-indexp 5 rex).
  (implies (and (syntaxp (quotep reg))
                (n03p reg))
           (reg-indexp reg rex)))

(encapsulate
 ()
 (local (include-book "arithmetic-5/top" :dir :system))

 (defun reg-index (reg rex name)

; Return the extension by one bit of the given 3-bit register index, in a
; context where the given name (b, x, r, or w) is supposed to determine the rex
; bit that provides the extension.
```

```
   (declare (xargs :guard (and (member-eq name '(b x r w))
                               (n08p rex)
                               (n03p reg))))
   (b* ((index (case name
                 (b 0)
                 (x 1)
                 (r 2)
                 (w 3))))
       (cond ((logbitp index rex)
              (logior 8 (mbe :logic (n03 reg)
                             :exec reg)))
             (t (mbe :logic (n03 reg)
                     :exec reg)))))

 (defthm reg-indexp-reg-index
   (reg-indexp (reg-index reg rex name) rex))

 (defthm reg-indexp-reg-index-type-prescription
   (natp (reg-index reg rex name))
   :rule-classes :type-prescription)

 (defthm reg-indexp-reg-index-linear
   (<= (reg-index reg rex name) 15)
   :rule-classes :linear)

 (defthm reg-indexp-forward
   (implies (reg-indexp reg rex)
            (n04p reg))
   :rule-classes :forward-chaining))

(in-theory (disable reg-index reg-indexp))

(defun 64-bit-modep (x86-64)
;;; @@ To be written
  (declare (xargs :guard (x86-64p x86-64)
                  :stobjs x86-64))
  (declare (ignore x86-64))
  t)

(defun rr08 (reg rex x86-64)

; Read a byte from a register.

  (declare (xargs :guard (and (n08p rex)
                              (reg-indexp reg rex)
                              (x86-64p x86-64)

; We probably don't really need the following guard -- certainly not for guard
; verification for this function -- but it serves as useful documentation and
; could catch an error when attempting guard verification for a caller.
```

```
                              (implies (rexp rex)
                                       (64-bit-modep x86-64)))
                  :stobjs x86-64))
  (cond ((or (rexp rex)
             (< reg 4))
         (let ((qword (rgfi reg x86-64)))
           (n08 qword)))
        (t ; no rex and Reg is at least 4 -- then read from AH etc.
         (let ((qword (rgfi (- reg 4) x86-64)))
           (n08 (ash qword -8))))))

(defthm n08p-rr08
  (n08p (rr08 reg rex x86-64))
  :rule-classes ((:type-prescription
                  :corollary
                  (natp (rr08 reg rex x86-64)))
                 (:linear
                  :corollary
                  (<= (rr08 reg rex x86-64) #xff))))

(in-theory (disable rr08))

(defun wr08 (reg byte rex x86-64)

; Write a byte to a register.

  (declare (xargs :guard (and (n08p byte)
                              (n08p rex)
                              (reg-indexp reg rex)
                              (x86-64p x86-64)
                              (implies (rexp rex)
                                       (64-bit-modep x86-64)))
                  :stobjs x86-64))
  (cond ((or (rexp rex)
             (< Reg 4))
         (let ((qword (rgfi Reg x86-64)))
           (!rgfi Reg
                  (logior (logand qword #uxffff_ffff_ffff_ff00)
                          byte)
                  x86-64)))
        (t ; no rex and Reg is at least 4 -- then write to AH etc.
         (let ((qword (rgfi (- Reg 4) x86-64)))
           (!rgfi Reg
                  (logior (logand qword #uxffff_ffff_ffff_00ff)
                          (ash byte 8))
                  x86-64)))))

(defthm x86-64p-wr08
  (implies (and (x86-64p x86-64)
                (reg-indexp reg rex)
                (n08p byte))
           (x86-64p (wr08 Reg byte rex x86-64))))

(in-theory (disable wr08))

(defun rr16 (reg x86-64)

; Read a word from a register.

  (declare (xargs :guard (and (n04p reg)
                              (x86-64p x86-64))
                  :stobjs x86-64))
  (n16 (rgfi reg x86-64)))

(defthm n16p-rr16
  (n16p (rr16 reg x86-64))
  :rule-classes ((:type-prescription
                  :corollary
                  (natp (rr16 reg x86-64)))
                 (:linear
                  :corollary
                  (<= (rr16 reg x86-64) #xffff))))

(in-theory (disable rr16))

(defun wr16 (reg val x86-64)

; Write a word to a register.

  (declare (xargs :guard (and (n16p val)
                              (n04p reg)
                              (x86-64p x86-64))
                  :stobjs x86-64))
  (let ((qword (rgfi Reg x86-64)))
    (!rgfi Reg
           (logior (logand qword #uxffff_ffff_ffff_0000)
                   val)
           x86-64)))

(defthm x86-64p-wr16
  (implies (and (x86-64p x86-64)
                (n16p val)
                (n04p reg))
           (x86-64p (wr16 reg val x86-64))))

(in-theory (disable wr16))

(defun rr32 (reg x86-64)

; Read a dword from a register.

  (declare (xargs :guard (and (n04p reg)
                              (x86-64p x86-64))
                  :stobjs x86-64))
  (n32 (rgfi reg x86-64)))

(defthm n32p-rr32
  (n32p (rr32 reg x86-64))
  :rule-classes ((:type-prescription
                  :corollary
                  (natp (rr32 reg x86-64)))
                 (:linear
                  :corollary
                  (<= (rr32 reg x86-64) #uxffff_ffff))))

(in-theory (disable rr32))

(defun wr32 (reg val x86-64)

; Write a dword to a register.  Note Intel Vol. 1 Sec. 3.4.1.1 p. 3-17, which
; says about 64-bit mode: "32-bit operands generate a 32-bit result,
; zero-extended to a 64-bit result in the destination general-purpose
; register."

; @@ Outside 64-bit mode, the upper 32 bits are undefined, as specified by the
; following quote from the same page as above:

;   Because the upper 32 bits of 64-bit general-purpose registers are undefined
;   in 32-bit modes, the upper 32 bits of any general-purpose register are not
;   preserved when switching from 64-bit mode to a 32-bit mode (to protected
;   mode or compatibility mode). Software must not depend on these bits to
;   maintain a value after a 64-bit to 32-bit mode switch.

; However, this function defines those bits to be 0.  At some point we should
; instead have this function make those bits as undefined, or perhaps if they
; are already known to be undefined (say, because we undefine them when moving
; out of 64-bit mode), then we leave them alone.

  (declare (xargs :guard (and (n32p val)
                              (n04p reg)
                              (x86-64p x86-64))
                  :stobjs x86-64))
  (!rgfi Reg
         val ; zero-extended to 64 bits; see above (Sec. 3.4.1.1 p. 3-17)
         x86-64))

(defthm x86-64p-wr32
  (implies (and (x86-64p x86-64)
                (n32p val)
                (n04p reg))
           (x86-64p (wr32 reg val x86-64))))

(in-theory (disable wr32))
```

```
(defun rr64 (reg x86-64)

; Read a qword from a register.

  (declare (xargs :guard (and (n04p reg)
                              (x86-64p x86-64))
                  :stobjs x86-64))
  (n64 (rgfi reg x86-64)))

(defthm n64p-rr64
  (n64p (rr64 reg x86-64))
  :rule-classes ((:type-prescription
                  :corollary
                  (natp (rr64 reg x86-64)))
                 (:linear
                  :corollary
                  (<= (rr64 reg x86-64) #uxffff_ffff_ffff_ffff))))

(in-theory (disable rr64))

(defun wr64 (reg val x86-64)

; Write a qword to a register.

  (declare (xargs :guard (and (n64p val)
                              (n04p reg)
                              (x86-64p x86-64))
                  :stobjs x86-64))
  (!rgfi Reg val x86-64))

(defthm x86-64p-wr64
  (implies (and (x86-64p x86-64)
                (n64p val)
                (n04p reg))
           (x86-64p (wr64 reg val x86-64))))

(in-theory (disable wr64))

(defun wm08? (addr byte x86-64)

; Write a byte to memory unless the address is out of range.

  (declare (type (unsigned-byte 64) addr)
           (type (unsigned-byte 8) byte)
           (xargs :guard (x86-64p x86-64)
                  :stobjs (x86-64)))
  (cond ((< addr *2^48*)
         (wm08 addr byte x86-64))
        (t
         (!ms (list (list 'wm08?
                         :addr addr
                         :byte byte))
             x86-64))))

(defthm x86-64p-wm08?
  (implies (and (x86-64p x86-64)
                (unsigned-byte-p 64 addr)
                (unsigned-byte-p 8 byte))
           (x86-64p (wm08? addr byte x86-64)))
  :hints (("Goal"
           :in-theory (enable x86-64p x86-64p-pre mem-array-next-addr))))

(in-theory (disable wm08?))

(defthm mv-nth-is-nth
  (equal (mv-nth i x)
         (nth i x))
  :hints (("Goal" :in-theory (enable nth))))

(defmacro get-instruction-bytes (itail ibytes n)

; Return (mv val itail ibytes), where val is the next n bytes of the
; instruction being decoded -- i.e., the low n bytes of itail -- and itail and
; ibytes are updated accordingly.

  (declare (xargs :guard (natp n)))
  (let* ((bits (ash n 3))
         (expt-2-bits (ash 1 bits))
         (mask (1- expt-2-bits)))
    `(let ((itail ,itail)
           (ibytes ,ibytes))
       (b* ((val (logand itail ,mask))
            (itail (ash itail ,(- bits)))
            (ibytes (+ ibytes ,n)))
           (mv val itail ibytes)))))

(defun get-instruction-byte (itail ibytes)

; Return (mv byte itail ibytes), where byte is the next byte of the instruction
; being decoded -- i.e., the low byte of itail -- and itail and ibytes are
; updated accordingly.

  (declare (xargs :guard (and (itailp itail)
                              (ibytesp ibytes))))
  (get-instruction-bytes itail ibytes 1))

(defthm n08p-nth-0-get-instruction-byte
  (implies (and (itailp itail)
                (ibytesp ibytes))
           (n08p (nth 0 (get-instruction-byte itail ibytes))))
  :rule-classes
  ((:type-prescription
    :corollary
    (implies (forced-and (itailp itail)
                         (ibytesp ibytes))
             (natp (nth 0 (get-instruction-byte itail ibytes)))))
   (:linear
    :corollary
    (implies (forced-and (itailp itail)
                         (ibytesp ibytes))
             (<= (nth 0 (get-instruction-byte itail ibytes))
                 #xff)))))

(defthm itailp-nth-1-get-instruction-byte
  (implies (forced-and (itailp itail)
                       (ibytesp ibytes))
           (itailp (nth 1 (get-instruction-byte itail ibytes))))
  :rule-classes :type-prescription)

(defthm ibytesp-nth-2-get-instruction-byte
  (implies (forced-and (itailp itail)
                       (ibytesp ibytes))
           (ibytesp (nth 2 (get-instruction-byte itail ibytes))))
  :rule-classes :type-prescription)

(in-theory (disable get-instruction-byte))

(defun get-instruction-word (itail ibytes)

; Return (mv word itail ibytes), where word is the next world of the
; instruction being decoded -- i.e., the low word of itail -- and itail and
; ibytes are updated accordingly.

  (declare (xargs :guard (and (itailp itail)
                              (ibytesp ibytes))))
  (get-instruction-bytes itail ibytes 2))

(defthm n16p-nth-0-get-instruction-word
  (implies (and (itailp itail)
                (ibytesp ibytes))
           (n16p (nth 0 (get-instruction-word itail ibytes))))
  :rule-classes
  ((:type-prescription
    :corollary
    (implies (forced-and (itailp itail)
                         (ibytesp ibytes))
             (natp (nth 0 (get-instruction-word itail ibytes)))))
   (:linear
    :corollary
    (implies (forced-and (itailp itail)
                         (ibytesp ibytes))
             (<= (nth 0 (get-instruction-word itail ibytes))
                 #xffff)))))

(defthm itailp-nth-1-get-instruction-word
  (implies (forced-and (itailp itail)
                       (ibytesp ibytes))
           (itailp (nth 1 (get-instruction-word itail ibytes))))
  :rule-classes :type-prescription)

(defthm ibytesp-nth-2-get-instruction-word
  (implies (forced-and (itailp itail)
                       (ibytesp ibytes))
           (ibytesp (nth 2 (get-instruction-word itail ibytes))))
  :rule-classes :type-prescription)

(in-theory (disable get-instruction-word))

(defun get-instruction-dword (itail ibytes)

; Return (mv dword itail ibytes), where dword is the next dword of the
; instruction being decoded -- i.e., the low dword of itail -- and itail and
; ibytes are updated accordingly.

  (declare (xargs :guard (and (itailp itail)
                              (ibytesp ibytes))))
  (get-instruction-bytes itail ibytes 4))

(defthm n32p-nth-0-get-instruction-dword
  (implies (and (itailp itail)
                (ibytesp ibytes))
           (n32p (nth 0 (get-instruction-dword itail ibytes))))
  :rule-classes
  ((:type-prescription
    :corollary
    (implies (forced-and (itailp itail)
                         (ibytesp ibytes))
             (natp (nth 0 (get-instruction-dword itail ibytes)))))
   (:linear
    :corollary
    (implies (forced-and (itailp itail)
                         (ibytesp ibytes))
             (<= (nth 0 (get-instruction-dword itail ibytes))
                 #xffffffff)))))

(defthm itailp-nth-1-get-instruction-dword
  (implies (forced-and (itailp itail)
                       (ibytesp ibytes))
           (itailp (nth 1 (get-instruction-dword itail ibytes))))
  :rule-classes :type-prescription)

(defthm ibytesp-nth-2-get-instruction-dword
  (implies (forced-and (itailp itail)
                      (ibytesp ibytes))
           (ibytesp (nth 2 (get-instruction-dword itail ibytes))))
  :rule-classes :type-prescription)

(in-theory (disable get-instruction-dword))

(defun get-instruction-qword (itail ibytes)

; Return (mv qword itail ibytes), where qword is the next qword of the
; instruction being decoded -- i.e., the low qword of itail -- and itail and
; ibytes are updated accordingly.

  (declare (xargs :guard (and (itailp itail)
                              (ibytesp ibytes))))
  (get-instruction-bytes itail ibytes 8))

(defthm n64p-nth-0-get-instruction-qword
  (implies (and (itailp itail)
                (ibytesp ibytes))
           (n64p (nth 0 (get-instruction-qword itail ibytes))))
  :rule-classes
  ((:type-prescription
    :corollary
    (implies (forced-and (itailp itail)
                         (ibytesp ibytes))
             (natp (nth 0 (get-instruction-qword itail ibytes)))))
   (:linear
    :corollary
    (implies (forced-and (itailp itail)
                         (ibytesp ibytes))
             (<= (nth 0 (get-instruction-qword itail ibytes))
                 #uxffffffff_ffffffff)))))

(defthm itailp-nth-1-get-instruction-qword
  (implies (forced-and (itailp itail)
                       (ibytesp ibytes))
           (itailp (nth 1 (get-instruction-qword itail ibytes))))
  :rule-classes :type-prescription)

(defthm ibytesp-nth-2-get-instruction-qword
  (implies (forced-and (itailp itail)
                       (ibytesp ibytes))
           (ibytesp (nth 2 (get-instruction-qword itail ibytes))))
  :rule-classes :type-prescription)

(in-theory (disable get-instruction-qword))

(defun operand-nbytes-p (nbytes)

; We would like to define this to be (member nbytes '(1 2 4 8)), but that
; imposes guard proof obligations that we can avoid (we hope, as of this
; writing) by using a simpler, more generous condition.

  (declare (xargs :guard t))
  (natp nbytes))

(defthm operand-nbytes-p-forward
  (implies (operand-nbytes-p x)
           (natp x))
  :rule-classes :forward-chaining)

(defun rgfi-size (nbytes index rex x86-64)
  (declare (xargs :guard (and (operand-nbytes-p nbytes)
                              (n08p rex)
                              (reg-indexp index rex)
                              (x86-64p x86-64))
                  :stobjs (x86-64)))
  (case nbytes
    (1 (rr08 index rex x86-64))
    (2 (rr16 index x86-64))
    (4 (rr32 index x86-64))
    (8 (rr64 index x86-64))
    (otherwise ; shouldn't happen
     0)))

(defun !rgfi-size (nbytes index val rex x86-64)

; We write only the indicated bytes (except for zero-extending 32-bit values in
; 64-bit mode; see wr32); the rest of the register is left unchanged, as
; suggested by Intel Vol. 2A p. 3-2 -- discussion of +rw etc. -- which
; references Table 3-1 on the next page.

  (declare (xargs :guard (and (operand-nbytes-p nbytes)
                              (bytesp nbytes val)
                              (n08p rex)
                              (reg-indexp index rex)
                              (x86-64p x86-64))
                  :stobjs (x86-64)))
  (case nbytes
    (1 (wr08 index val rex x86-64))
    (2 (wr16 index val x86-64))
    (4 (wr32 index val x86-64))
    (8 (wr64 index val x86-64))
    (otherwise ; shouldn't happen
     x86-64)))

(defun rm-size (nbytes addr x86-64)
  (declare (xargs :guard (and (operand-nbytes-p nbytes)
                              (natp addr)
                              (<= (+ addr nbytes) *2^48*)
                              (x86-64p x86-64))
                  :stobjs (x86-64)))
  (case nbytes
    (1 (rm08 addr x86-64))
    (2 (rm16 addr x86-64))
    (4 (rm32 addr x86-64))
    (8 (rm64 addr x86-64))
    (otherwise ; shouldn't happen
     0)))

(defun wm-size (nbytes addr val x86-64)
  (declare (xargs :guard (and (operand-nbytes-p nbytes)
                              (natp addr)
                              (<= (+ addr nbytes) *2^48*)
                              (bytesp nbytes val)
                              (x86-64p x86-64))
                  :stobjs (x86-64)))
  (case nbytes
    (1 (wm08 addr val x86-64))
    (2 (wm16 addr val x86-64))
    (4 (wm32 addr val x86-64))
    (8 (wm64 addr val x86-64))
    (otherwise ; shouldn't happen
     x86-64)))

; Step function support

(defun x86-64-step-name (op-name)
  (declare (xargs :guard (symbolp op-name)))
  (packn (list 'x86-64-step- op-name)))

(defconst *decoded-fields*
  '(p1 p2 p3 p4 rex opcode ModR/M sib displacement
       immediate ; now comes extra info from the parser:
       operand-nbytes ModR/M-p))

(defun op-alist-value-p (x)
  (declare (xargs :guard t))
  (cond ((atom x) (null x))
        (t (and (let ((e (car x)))
                  (or (natp e)
                      (and (consp e)
                           (natp (car e))
                           (natp (cdr e)))))
                (op-alist-value-p (cdr x))))))

(defun op-alistp (x)
  (declare (xargs :guard t))
  (cond ((atom x)
         (null x))
        ((atom (car x))
         nil)
        (t (and (symbolp (caar x))
                (op-alist-value-p (cdar x))
                (op-alistp (cdr x))))))

(defun op-alist-value-opcodes (val)
  (declare (xargs :guard (op-alist-value-p val)))
  (cond ((endp val) nil)
        (t (let* ((x (car val))
                  (opcode (if (consp x) (car x) x)))
             (add-to-set opcode
                         (op-alist-value-opcodes (cdr val)))))))

(defun not-equal-opcode-lst (opcodes var)
  (declare (xargs :guard (nat-listp opcodes)))
  (cond ((endp opcodes) nil)
        (t (cons `(not (equal ,var ,(car opcodes)))
                 (not-equal-opcode-lst (cdr opcodes) var)))))

(defun defopcodes-fn (op-alist acc)
; See defopcodes for a warning.
  (declare (xargs :guard (and (op-alistp op-alist)
                              (true-listp acc))))
  (cond ((endp op-alist)
         (cons 'progn (reverse acc)))
        (t (defopcodes-fn
             (cdr op-alist)
             (append (let* ((entry (car op-alist))
                            (name (car entry))
                            (val (cdr entry))
                            (opcodes (op-alist-value-opcodes val))
                            (name$opcodep (packn (list name "$OPCODEP")))
                            (name$opcodep-false
                             (packn (list name "$OPCODEP-FALSE"))))
                       `((in-theory (disable ,name$opcodep))
                         (defthm ,name$opcodep-false
                           (implies (and ,@(not-equal-opcode-lst opcodes 'x))
                                    (equal (,name$opcodep x)
                                           nil))
                           :rule-classes ((:forward-chaining
                                           :trigger-terms ((,name$opcodep x)))))
                         (defun ,name$opcodep (x)
                           (declare (xargs :guard t))
                           (member-equal x ',opcodes))))
                     acc)))))

(defun get-ints (x)
  (declare (xargs :guard (true-listp x)))
  (cond ((endp x) nil)
        ((integerp (car x))
         (cons (car x) (get-ints (cdr x))))
        (t (get-ints (cdr x)))))
```

```
(defun x86-64-step-cases-fn-1 (op-alist acc)
  (declare (xargs :guard (and (op-alistp op-alist)
                              (true-listp acc))))
  (cond ((endp op-alist) acc)
        (t (x86-64-step-cases-fn-1
            (cdr op-alist)
            (cons (let* ((entry (car op-alist))
                         (op-name (car entry))
                         (step-name (x86-64-step-name op-name))
                         (opcodes (get-ints (cdr entry))))
                    `(,opcodes
                      (,step-name ,@*decoded-fields* ibytes x86-64)))
                  acc)))))

(defun op-alist-to-op-pairs-alist-1 (sym op-alist-value op-pairs-alist)
  (declare (xargs :guard (and (symbolp sym)
                              (op-alist-value-p op-alist-value)
                              (eqlable-alistp op-pairs-alist))))
  (cond ((endp op-alist-value) op-pairs-alist)
        ((integerp (car op-alist-value))
         (op-alist-to-op-pairs-alist-1 sym (cdr op-alist-value)
                                       op-pairs-alist))
        (t (let ((key (caar op-alist-value)))
             (op-alist-to-op-pairs-alist-1
              sym
              (cdr op-alist-value)
              (let ((old (assoc key op-pairs-alist)))
                (cond (old (put-assoc key
                                      (cons (cons sym (cdar op-alist-value))
                                            (cdr old))
                                      op-pairs-alist))
                      (t (acons key
                                (list (cons sym (cdar op-alist-value)))
                                op-pairs-alist)))))))))

(defthm eqlable-alistp-put-assoc-equal
  (implies (and (eqlable-alistp x)
                (eqlablep key))
           (eqlable-alistp (put-assoc-equal key val x))))

(defun op-alist-to-op-pairs-alist (op-alist op-pairs-alist)

; Maps an op-alist (see e.g. *op-alist* in x86.lisp) to an alist associating
; numbers with lists of pairs (sym . 3-bit-number).  For example, for an early
; version of *op-alist* we have:
;   ACL2 !>(op-alist-to-op-pairs-alist *op-alist* nil)
;   ((255 (PUSH . 6) (INC . 0))
;    (254 (INC . 0)))
;   ACL2 !>

  (declare (xargs :guard (and (op-alistp op-alist)
                             (eqlable-alistp op-pairs-alist))))
  (cond ((endp op-alist) op-pairs-alist)
        (t (op-alist-to-op-pairs-alist
            (cdr op-alist)
            (op-alist-to-op-pairs-alist-1 (caar op-alist)
                                          (cdar op-alist)
                                          op-pairs-alist)))))

(defun op-pairs-listp (x)
  (declare (xargs :guard t))
  (cond ((atom x) (null x))
        (t (and (consp (car x))
                (symbolp (caar x))
                (integerp (cdar x))
                (op-pairs-listp (cdr x))))))

(defun op-pairs-alistp (x)
  (declare (xargs :guard t))
  (cond ((atom x) (null x))
        (t (and (consp (car x))
                (integerp (caar x))
                (op-pairs-listp (cdar x))
                (op-pairs-alistp (cdr x))))))

(defthm op-pairs-alistp-put-assoc-equal
  (implies (and (integerp key)
                (op-pairs-listp x)
                (op-pairs-alistp op-pairs-alist))
           (op-pairs-alistp
            (put-assoc-equal key x op-pairs-alist))))

(defthm op-pairs-listp-cdr-assoc-equal
  (implies (op-pairs-alistp op-pairs-alist)
           (op-pairs-listp (cdr (assoc-equal key op-pairs-alist)))))

(defthm op-pairs-alistp-op-alist-to-op-pairs-alist-1
  (implies (and (symbolp sym)
                (op-alist-value-p op-alist-value)
                (op-pairs-alistp op-pairs-alist))
           (op-pairs-alistp
            (op-alist-to-op-pairs-alist-1 sym op-alist-value op-pairs-alist))))

(defthm op-pairs-alistp-op-alist-to-op-pairs-alist
  (implies (and (op-alistp op-alist)
                (op-pairs-alistp op-pairs-alist))
           (op-pairs-alistp
            (op-alist-to-op-pairs-alist op-alist op-pairs-alist))))

(defun x86-64-step-cases-fn-2-1 (pairs)
  (declare (xargs :guard (op-pairs-listp pairs)))
  (cond ((endp pairs)
         '((otherwise (x86-64-step-unimplemented opcode x86-64))))
        (t (cons (let ((step-name (x86-64-step-name (caar pairs)))
                       (ext (cdar pairs)))
                   `(,ext
                     (,step-name ,@*decoded-fields* ibytes x86-64)))
                 (x86-64-step-cases-fn-2-1 (cdr pairs))))))

(defun x86-64-step-cases-fn-2 (op-pairs-alist acc)
  (declare (xargs :guard (and (op-pairs-alistp op-pairs-alist)
                              (true-listp acc))))
  (cond ((endp op-pairs-alist) acc)
        (t (x86-64-step-cases-fn-2
            (cdr op-pairs-alist)
            (cons (let* ((entry (car op-pairs-alist))
                         (opcode (car entry))
                         (op-pairs (cdr entry)))
                    `(,opcode
                      (case (mrm-reg ModR/M)
                        ,@(x86-64-step-cases-fn-2-1 op-pairs))))
                  acc)))))

(defun x86-64-step-cases-fn (op-alist acc)
  (let* ((acc1 (x86-64-step-cases-fn-2 (op-alist-to-op-pairs-alist op-alist
                                                                    nil)
                                       acc))
         (acc2 (x86-64-step-cases-fn-1 op-alist acc1)))
    `(case opcode ,@acc2)))

(defun sign-extend (x from to)
  (declare (xargs :guard (and (integerp x) (posp from) (integerp to))))
  (let ((neg-p (logbitp (1- from) x)))
    (if neg-p
        (logand (- x (ash 1 from))
                (1- (ash 1 to)))
      x)))
```

15 ========== File x86-64/x86-general.lisp ==========

```
(in-package "ACL2")

(include-book "x86-utils")

; We separate out instruction decoding based purely on the 15 bytes.  (Useful
; later for debugging.)

;  "Table of contents":
;   1. Get 15 bytes (will become much more complicated with memory management),
;      checking read+execute permissions.
;   2. Return instruction fields using mv.  This is where we cause an error if
;      we find we need more than 15 bytes.
;   3. Get the effective addresses for memory accesses.
;   4. Check read and write permissions.
;   5. Branch on opcode and call the individual instruction update functions.

; OK, let's start!

;   1. Get 15 bytes (will become much more complicated with memory management),
;      checking read+execute permissions

(defun x86-64-fetch (x86-64)
; Returns (mv erp n), where n is 15 bytes if erp is nil.
  (declare (xargs :guard (x86-64p x86-64)
                  :stobjs (x86-64)))
  (b* ((ctx 'x86-64-fetch)
       (rip (rip x86-64))
       ((when (>= (+ rip 16)

; The use of 16 above is conservative; reading 15 bytes would suffice for x86
; semantics, but since we want to use rm128 to do our read, we use 16.

                  *2^48*))
        (mv (!ms-erp-fresh :rip rip)
            0)))
      (mv nil

; The approach just below is inefficient in the sense that we create two
; 2-quadword numbers in CCL for the two 64-bit reads, and then a 3 quadword
; number in the result, and then another 2-quadword number by the logand.  But
; at this point we opt for clarity rather than efficiency.

          (logand (rm128 rip x86-64)
                  #ux00ffffff_ffffffff_ffffffff_ffffffff))))

(defthm n120p-nth-1-x86-64-fetch
  (implies (x86-64p x86-64)
           (n120p (nth 1 (x86-64-fetch x86-64))))
  :rule-classes
  ((:type-prescription
    :corollary
    (implies (force (x86-64p x86-64))
             (natp (nth 1 (x86-64-fetch x86-64)))))
   (:linear
    :corollary
    (implies (force (x86-64p x86-64))
             (<= (nth 1 (x86-64-fetch x86-64))
                 #ux00ffffff_ffffffff_ffffffff_ffffffff)))))

(in-theory (disable x86-64-fetch))

;   2. Return instruction fields using mv.  This is where we cause an error if
;      we find we need more than 15 bytes.

(defconst *x86-64-prefixes*
  '( ;; Group 1
    ;; Bus lock
    (#xF0 1 lock)
    ;; Repeat-not-zero prefix; for string and input/output
    (#xF2 1 repne repnz)
    ;; Repeat-zero prefix; for string and input/output
    (#xF3 1 rep repe repz)

    ;; Group 2
    ;; Segment overrides (Intel Vol. 2 says: "use with any branch instruction
    ;; is reserved")
    (#x2E 2 cs-override cs) ; Ignored in 64-bit mode
    (#x36 2 ss-override ss) ; Ignored in 64-bit mode
    (#x3E 2 ds-override ds) ; Ignored in 64-bit mode
    (#x26 2 es-override es) ; Ignored in 64-bit mode
    (#x64 2 fs-override)
    (#x65 2 gs-override)
    ;; Branch hints (Intel Vol. 2 says: "used only with Jcc instructions")
    (#x2E 2 branch-not-taken bnt)
    (#x3E 2 branch-taken bt)

    ;; Group 3
    ;; Operand-size override
    (#x66 3 operand-override op-override op)

    ;; Group 4
    ;; Address-size override
    (#x67 4 address-override addr-override addr)))

(defun x86-64-decode-prefix-rec (itail ibytes p1 p2 p3 p4)
; Returns (mv erp p1 p2 p3 p4 itail ibytes).
  (declare (xargs :guard (and (itailp itail)
                              (ibytesp ibytes))
                  :measure (- 4 (+ (if p1 1 0)
                                   (if p2 1 0)
                                   (if p3 1 0)
                                   (if p4 1 0)))))
  (b* ((ctx 'x86-64-decode-prefix)
       (byte (n08 itail))
       (entry (assoc byte *x86-64-prefixes*)))
      (cond ((null entry)
             (mv nil p1 p2 p3 p4 itail ibytes))
            (t (case (cadr entry)
                 (1 (cond (p1 (mv (!ms-erp-fresh :p1-duplicates (list p1 byte))
                                  p1 p2 p3 p4 itail ibytes))
                          (t (x86-64-decode-prefix-rec
                              (ash itail -8) (1+ ibytes) byte p2 p3 p4))))
                 (2 (cond (p2 (mv (!ms-erp-fresh :p2-duplicates (list p2 byte))
                                  p1 p2 p3 p4 itail ibytes))
                          (t (x86-64-decode-prefix-rec
                              (ash itail -8) (1+ ibytes) p1 byte p3 p4))))
                 (3 (cond (p3 (mv (!ms-erp-fresh :p3-duplicates (list p3 byte))
                                  p1 p2 p3 p4 itail ibytes))
                          (t (x86-64-decode-prefix-rec
                              (ash itail -8) (1+ ibytes) p1 p2 byte p4))))
                 (4 (cond (p4 (mv (!ms-erp-fresh :p4-duplicates (list p4 byte))
                                  p1 p2 p3 p4 itail ibytes))
                          (t (x86-64-decode-prefix-rec
                              (ash itail -8) (1+ ibytes) p1 p2 p3 byte))))
                 (otherwise (mv (er hard ctx
                                    "Unexpected case, ~x0"
                                    (cadr entry))
                                p1 p2 p3 p4 itail ibytes)))))))

(defun x86-64-decode-prefix (itail ibytes)
; Returns (mv erp p1 p2 p3 p4 itail ibytes).
  (declare (xargs :guard (and (itailp itail)
                              (ibytesp ibytes))))
  (x86-64-decode-prefix-rec itail ibytes nil nil nil nil))

(defun x86-64-decode-rex (itail ibytes 64-bit-modep)
; Returns (mv rex itail ibytes), where rex is a byte.
  (declare (xargs :guard (and (itailp itail)
                              (ibytesp ibytes))))
  (cond ((not 64-bit-modep)
         (mv 0 itail ibytes))
        ((int= (logand #xf0 itail) #x40)
         (get-instruction-byte itail ibytes))
        (t (mv 0 itail ibytes))))

(defun x86-64-decode-opcode (itail ibytes)

; Returns (mv opcode itail ibytes), where opcode is an opcodep.

; See Intel Manual A.4.  Here we do not include the possible extension formed
; using 3 bits from the ModR/M byte.
```

```
  (declare (xargs :guard (and (itailp itail)
                              (ibytesp ibytes))))
;;; @@ Need to generalize past 1-byte case.
  (get-instruction-byte itail ibytes))

(defun opcodep (x)

; We are thinking of an opcode as seems to be suggested in the AMD manual: 1
; byte, without any 3-bit extension (as that register index will simply be
; incorporated into the semantics).  @@ At this point, we are handling only
; 1-byte opcodes.  Later, we will probably leave this function alone but pass
; around an indicator of which opcode table we are using.

  (declare (xargs :guard t))
  (n08p x))

(defthm opcodep-forward
  (implies (opcodep x)
           (n08p x))
  :rule-classes :forward-chaining)

(defthm opcodep-backward
  (implies (n08p x)
           (opcodep x)))

(in-theory (disable opcodep))

(defund x86-64-decode-ModR/M-p (opcode)
; Returns a Boolean saying whether the given opcode expects a ModR/M byte.
  (declare (xargs :guard (opcodep opcode)))
  (aref1 'onebyte-has-modrm *onebyte-has-modrm-ar* opcode))

(defmacro mrm-r/m (ModR/M)
  `(n03 ,ModR/M))

(defmacro mrm-mod (ModR/M)
  `(ash ,ModR/M -6))

(defmacro mrm-reg (ModR/M)
  `(n03 (ash ,ModR/M -3)))

(defun x86-64-decode-sib (itail ibytes ModR/M)
; Returns (mv sib itail ibytes), where sib is nil or a byte.
  (declare (xargs :guard (and (itailp itail)
                              (ibytesp ibytes)
                              (n08p ModR/M))))
  (b* ((r/m (mrm-r/m ModR/M)))
      (cond ((and (int= r/m 4)
                  (not (int= (mrm-mod ModR/M) 3))) ; not register-to-register
             (get-instruction-byte itail ibytes))
```

```
            (t (mv nil itail ibytes)))))

(defun x86-64-decode-displacement (itail ibytes ModR/M)

; Returns (mv displacement itail ibytes), where displacement is a (possibly
; negative) integer.

; We sign-extend displacements to 64 bits in 64-bit mode (Intel Sec. 2.2.1.5 on
; p. 2-14.  How about other modes?  Footnote 3 on Intel p. 2-7 settles this for
; displacements: sign-extend 8-bit displacements.  Thus, this function returns
; a signed integer.  We'll leave it to the consumer of the resulting integer
; displacement to truncate as appropriate.

  (declare (xargs :guard (and (itailp itail)
                              (ibytesp ibytes)
                              (n08p ModR/M))))
  (b* ((r/m (mrm-r/m ModR/M))
       (mod (mrm-mod ModR/M)))
      (case mod
        (0 (case r/m
             (5 (b* (((mv dword itail ibytes)
                      (get-instruction-dword itail ibytes)))
                    (mv (n32-to-i32 dword) itail ibytes)))
             (otherwise (mv 0 itail ibytes))))
        (1 (b* (((mv byte itail ibytes)
                 (get-instruction-byte itail ibytes)))
               (mv (n08-to-i08 byte) itail ibytes)))
        (2 (b* (((mv dword itail ibytes)
                 (get-instruction-dword itail ibytes)))
               (mv (n32-to-i32 dword) itail ibytes)))
        (otherwise ; 3 (not relevant)
         (mv 0 itail ibytes)))))

(defun opcode-array-element-p (x)
  (declare (xargs :guard t))
  (or (eq x 'x)
      (eq x '-)
      (natp x)))

(defun opcode-array-element-listp (x)
  (declare (xargs :guard (true-listp x)))
  (cond ((endp x) t)
        (t (and (opcode-array-element-p (car x))
                (opcode-array-element-listp (cdr x))))))

(defun opcode-array-entryp (x)
  (declare (xargs :guard t))
  (or (opcode-array-element-p x)
      (and (true-listp x)
           (equal (length x) 8)
           (opcode-array-element-listp x))))
```

```
(defun opcode-array-p1 (i name ar)
  (declare (xargs :guard (and (array1p name ar)
                              (natp i)
                              (<= i (car (dimensions name ar))))))
  (cond ((zp i) t)
        (t (let ((i (1- i)))
             (and (opcode-array-entryp (aref1 name ar i))
                  (opcode-array-p1 i name ar))))))

(defthm opcode-array-p1-element-type-lemma
  (implies (and (natp i)
                (<= i 256)
                (opcode-array-p1 i name ar)
                (< j i)
                (natp j)
                (not (opcode-array-element-p (cdr (assoc-equal j ar)))))
           (true-listp (cdr (assoc-equal j ar))))
  :rule-classes nil)

(in-theory (disable opcode-array-p1))

(defthm opcode-array-p1-element-type
  (implies (and (not (true-listp (cdr (assoc-equal j ar))))
                (natp j)
                (< j 256)
                (not (opcode-array-element-p (cdr (assoc-equal j ar)))))
           (not (opcode-array-p1 256 name ar)))
  :hints (("Goal" :use ((:instance opcode-array-p1-element-type-lemma
                                   (i 256))))))

(defun opcode-array-p (name ar)
  (declare (xargs :guard t))
  (and (array1p name ar)
       (equal (dimensions name ar)
              '(256))
       (equal (default name ar) 'x)
       (opcode-array-p1 256 name ar)))

(in-theory (disable array1p dimensions default))

(defun x86-64-decode-immediate (itail ibytes operand-nbytes opcode
                                      decode-immediate-ar)

; Returns (mv immediate ibytes), where we expect (bytesp operand-nbytes
; immediate); see decoded-instruction-p.

; According to Intel Vol. 2 Sec. 3.1.1.3 p. 3-7, immediates are to be viewed as
; signed numbers.  We leave it to the caller to do any necessary conversion of
; the fixed-width immediate value returned (a natural number) to a (possibly
; signed) integer.
; Note that 64-bit immediates are legal with REX for a MOV instruction
; targeting a GPR; see Intel 2.2.1.5, p. 2-14, and see AMD Vol. 3 Sec. 1.6
; (p. 24), which restricts 64-bit immediates to "MOV instructions that load
; GPRs".  And see AMD Vol. 1 Sec. 3.2.3.3 ("Immediate Operand Size"), which
; specifies those opcodes as "B8h through BFh".

; Unlike the other decode-xxx functions, this one does not return an itail
; (because the caller doesn't need that).

  (declare (xargs :guard (and (itailp itail)
                              (ibytesp ibytes)
                              (natp operand-nbytes)
                              (opcodep opcode)
                              (opcode-array-p 'decode-immediate
                                              decode-immediate-ar))))
; @@ So far we only handle 1-byte opcodes.
  (cond
   ((int= operand-nbytes 0) ; signifies no immediate
; We added this extra case for proving x86-64-decode-immediate-correctness.
    (mv 0 ibytes))
   (t
    (case (aref1 'decode-immediate decode-immediate-ar opcode)
      (0 ; no immediate bytes
       (mv 0 ibytes))
      (1 ; one immediate byte
       (mv (n08 itail) (1+ ibytes)))
      (2 ; grab operand-nbytes many bytes for immediate

; AMD Vol. 1 p. 43: "In 64-bit mode, if the operand size is 64 bits (requires a
; REX prefix), these instructions can be used to copy a true 64-bit immediate
; into a GPR."

; Note: We use this case for instructions like call instruction #xe8, where
; there are no operands and hence operand-bytes refers to the size of the
; immediate.

       (mv (nx (ash operand-nbytes 3) itail)
           (+ ibytes operand-nbytes)))
      (3

; This is a special case, for #xc7 and others for which a 64-bit operand calls
; for only a 32-bit immediate.

       (let ((nbytes (min operand-nbytes 4)))
         (mv (nx (ash nbytes 3) itail)
             (+ ibytes nbytes))))
      (otherwise ; @@ don't-care, but maybe should cause an error
       (mv 0 ibytes))))))

(defund rex-wp (rex)
; Returns a Boolean for the W bit of rex (see Intel Vol. 2 Table 2-4 p. 2-11).
  (declare (xargs :guard (n08p rex)))
  (not (zerop (logand rex #b1000))))

(defund rex-rp (rex)
; Returns a Boolean for the R bit of rex (see Intel Vol. 2 Table 2-4 p. 2-11).
  (declare (xargs :guard (n08p rex)))
  (not (zerop (logand rex #b100))))

(defund rex-xp (rex)
; Returns a Boolean for the X bit of rex (see Intel Vol. 2 Table 2-4 p. 2-11).
  (declare (xargs :guard (n08p rex)))
  (not (zerop (logand rex #b10))))

(defund rex-bp (rex)
; Returns a Boolean for the B bit of rex (see Intel Vol. 2 Table 2-4 p. 2-11).
  (declare (xargs :guard (n08p rex)))
  (not (zerop (logand rex #b1))))

(defund cs-dp (x86-64)

; Returns a Boolean.  See Intel Vol. 1 Table 3-3 p. 3-25.

  (declare (xargs :guard (x86-64p x86-64)
                  :stobjs (x86-64)))
; We started writing this function based on the segment registers and segment
; selectors, and then realized that it might be better just to use fields of
; the x86-64 stobj that are currently commented out (seg-base, seg-limit, and
; seg-access).  So for now, we'll restrict to the spirit of 64-bit mode and
; simply return t.
#||
  (b* ((cs-posn (m86-reg-pos :cs *m86-64-segment-reg-names*))
       (cs (segi cs-posn x86-64))
       (index (x86-segment-selector :index cs)) ; Intel Vol. 3 Fig. 3-6 p. 3-10
       (offset (ash index 3))
       (base
||#
  (declare (ignore x86-64))
  t)

; Because defund doesn't disable the type-prescription rule, which in this case
; is very powerful:
(in-theory (disable (:type-prescription cs-dp)))

(defund opcode-array-element (name ar opcode opcode-ext)
  (declare (xargs :guard (and (opcode-array-p name ar)
                              (opcodep opcode)
                              (n03p opcode-ext))))
  (let ((val (aref1 name ar opcode)))
    (cond ((atom val) val)
          (t (nth opcode-ext val)))))
```

```
(defun x86-64-decode-operand-nbytes (p3 rex opcode opcode-ext cs-dp
                                        decode-operand-nbytes-ar)

; Returns the number of bytes in operands for the given opcode (with the given
; prefixes etc.).  Note that "operands" also includes those that come from
; immediates, not (for example) only those that come from the ModR/M.  If there
; are no operands, the returned number of bytes should be the number of bytes
; in the immediate.  If there are both an immediate and at least one operand,
; then return the number of bytes in the operand.

; See Intel Sec. 2.2.1.2 p. 2-10.

  (declare (xargs :guard (and (n08p? p3)
                              (n08p rex)
                              (opcodep opcode)
                              (n03p opcode-ext)
                              (opcode-array-p 'decode-operand-nbytes
                                              decode-operand-nbytes-ar))))
  (case (opcode-array-element 'decode-operand-nbytes
                              decode-operand-nbytes-ar
                              opcode opcode-ext)
    (0 0) ; no operands and no immediate
    (1 1) ; one byte
    (2

; @@ 64-bit mode only?

     (if (rex-wp rex)
         8
       (if cs-dp ; see Intel Vol. 1 Table 3-3 p. 3-25
           (if p3 2 4)
         (if p3 4 2))))
    (3 ; @@ 64-bit mode only

; This case handles 64-bit mode instructions that implicitly reference the RSP,
; as well as near branches.

; Intel Vol. A Sec. 2.2.1.2 p. 2-10: "For non-byte operations: if a 66H prefix
; is used with prefix (REX.W = 1), 66H is ignored."

; Intel Vol. A Sec. 2.2.1.7 p. 2-15, section "Default 64-Bit Operand Size",
; says:

;    In 64-bit mode, two groups of instructions have a default operand size of
;    64 bits (do not need a REX prefix for this operand size). These are:

;    * Near branches
;    * All instructions, except far branches, that implicitly reference the
;      RSP.
; Also note AMD Vol. 3 p. 253 (documentation for PUSH): "In 64-bit mode, the
; operand size of all PUSH instructions defaults to 64 bits, and there is no
; prefix available to encode a 32-bit operand size."

     (if (rex-wp rex)
         8
       (if cs-dp ; see Intel Vol. 1 Table 3-3 p. 3-25
           (if p3 2 8)
         (if p3 8 2))))
    (4

; We can use this for instructions like the call instruction, #xe8, where there
; aren't any operands and hence we are referring to the number of bytes in the
; immediate.

     4)
    (otherwise ; @@ stub
     4)))

(encapsulate
 ()

 (local
  (defthm decode-prefix-rec-correctness
    (let* ((dec (x86-64-decode-prefix-rec itail0 ibytes0 p10 p20 p30 p40))
           (p1 (nth 1 dec))
           (p2 (nth 2 dec))
           (p3 (nth 3 dec))
           (p4 (nth 4 dec))
           (itail (nth 5 dec))
           (ibytes (nth 6 dec)))
      (implies (and (natp itail0)
                    (natp ibytes0)
                    (n08p? p10)
                    (n08p? p20)
                    (n08p? p30)
                    (n08p? p40))
               (and (implies p1
                             (n08p p1))
                    (implies p2
                             (n08p p2))
                    (implies p3
                             (n08p p3))
                    (implies p4
                             (n08p p4))
                    (natp itail)
                    (natp ibytes))))
    :hints (("Goal" :in-theory (disable assoc)))
    :rule-classes
    (:rewrite
     (:linear
      :corollary
      (implies (and (n08p? p10)
                    (n08p? p20)
                    (n08p? p30)
                    (n08p? p40)
                    (natp itail0)
                    (natp ibytes0))
               (<= (nth 1
                        (x86-64-decode-prefix-rec itail0 ibytes0 p10 p20 p30 p40))
                   255)))
     (:linear
      :corollary
      (implies (and (n08p? p10)
                    (n08p? p20)
                    (n08p? p30)
                    (n08p? p40)
                    (natp itail0)
                    (natp ibytes0))
               (<= (nth 2
                        (x86-64-decode-prefix-rec itail0 ibytes0 p10 p20 p30 p40))
                   255)))
     (:linear
      :corollary
      (implies (and (n08p? p10)
                    (n08p? p20)
                    (n08p? p30)
                    (n08p? p40)
                    (natp itail0)
                    (natp ibytes0))
               (<= (nth 3
                        (x86-64-decode-prefix-rec itail0 ibytes0 p10 p20 p30 p40))
                   255)))
     (:linear
      :corollary
      (implies (and (n08p? p10)
                    (n08p? p20)
                    (n08p? p30)
                    (n08p? p40)
                    (natp itail0)
                    (natp ibytes0))
               (<= (nth 4
                        (x86-64-decode-prefix-rec itail0 ibytes0 p10 p20 p30 p40))
                   255))))))

 (defthm decode-prefix-correctness
   (let* ((dec (x86-64-decode-prefix itail0 ibytes0))
          (p1 (nth 1 dec))
          (p2 (nth 2 dec))
          (p3 (nth 3 dec))
          (p4 (nth 4 dec))
          (itail (nth 5 dec))
```

```
       (ibytes (nth 6 dec)))
  (implies (and (natp itail0)
                (natp ibytes0))
           (and (implies p1
                         (n08p p1))
                (implies p2
                         (n08p p2))
                (implies p3
                         (n08p p3))
                (implies p4
                         (n08p p4))
                (natp itail)
                (natp ibytes))))
:rule-classes
((:type-prescription
  :corollary
  (implies (forced-and (natp itail0)
                       (natp ibytes0))
           (natp? (nth 1 (x86-64-decode-prefix itail0 ibytes0)))))
 (:type-prescription
  :corollary
  (implies (forced-and (natp itail0)
                       (natp ibytes0))
           (natp? (nth 2 (x86-64-decode-prefix itail0 ibytes0)))))
 (:type-prescription
  :corollary
  (implies (forced-and (natp itail0)
                       (natp ibytes0))
           (natp? (nth 3 (x86-64-decode-prefix itail0 ibytes0)))))
 (:type-prescription
  :corollary
  (implies (forced-and (natp itail0)
                       (natp ibytes0))
           (natp? (nth 4 (x86-64-decode-prefix itail0 ibytes0)))))
 (:type-prescription
  :corollary
  (implies (forced-and (natp itail0)
                       (natp ibytes0))
           (natp (nth 5 (x86-64-decode-prefix itail0 ibytes0)))))
 (:type-prescription
  :corollary
  (implies (forced-and (natp itail0)
                       (natp ibytes0))
           (natp (nth 6 (x86-64-decode-prefix itail0 ibytes0)))))
 (:linear
  :corollary
  (implies (forced-and (natp itail0)
                       (natp ibytes0))
           (<= (nth 1 (x86-64-decode-prefix itail0 ibytes0))
               #xff))
  :hints (("Goal"
               :cases
               ((nth 1 (x86-64-decode-prefix-rec itail0 ibytes0 nil nil nil nil))))))
     (:linear
      :corollary
      (implies (forced-and (natp itail0)
                           (natp ibytes0))
               (<= (nth 2 (x86-64-decode-prefix itail0 ibytes0))
                   #xff)))
     (:linear
      :corollary
      (implies (forced-and (natp itail0)
                           (natp ibytes0))
               (<= (nth 3 (x86-64-decode-prefix itail0 ibytes0))
                   #xff)))
     (:linear
      :corollary
      (implies (forced-and (natp itail0)
                           (natp ibytes0))
               (<= (nth 4 (x86-64-decode-prefix itail0 ibytes0))
                   #xff)))))

(defthm decode-rex-correctness
  (let* ((dec (x86-64-decode-rex itail0 ibytes0 64-bit-modep))
         (rex (nth 0 dec))
         (itail (nth 1 dec))
         (ibytes (nth 2 dec)))
    (implies (and (natp itail0)
                  (natp ibytes0))
             (and (n08p rex)
                  (natp itail)
                  (natp ibytes))))
  :rule-classes
  ((:type-prescription
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (natp (nth 0 (x86-64-decode-rex itail0 ibytes0 64-bit-modep)))))
   (:type-prescription
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (natp (nth 1 (x86-64-decode-rex itail0 ibytes0 64-bit-modep)))))
   (:type-prescription
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (natp (nth 2 (x86-64-decode-rex itail0 ibytes0 64-bit-modep)))))
   (:linear
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (<= (nth 0 (x86-64-decode-rex itail0 ibytes0 64-bit-modep))
                 #xff)))))

(defthm get-instruction-byte-correctness
  (let* ((dec (get-instruction-byte itail0 ibytes0))
         (byte (nth 0 dec))
         (itail (nth 1 dec))
         (ibytes (nth 2 dec)))
    (implies (and (natp itail0)
                  (natp ibytes0))
             (and (n08p byte)
                  (natp itail)
                  (natp ibytes))))
  :rule-classes
  ((:type-prescription
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (natp (nth 0 (get-instruction-byte itail0 ibytes0)))))
   (:type-prescription
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (natp (nth 1 (get-instruction-byte itail0 ibytes0)))))
   (:type-prescription
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (natp (nth 2 (get-instruction-byte itail0 ibytes0)))))
   (:linear
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (<= (nth 0 (get-instruction-byte itail0 ibytes0))
                 #xff)))))

(defthm x86-64-decode-opcode-correctness
  (let* ((dec (x86-64-decode-opcode itail0 ibytes0))
         (byte (nth 0 dec))
         (itail (nth 1 dec))
         (ibytes (nth 2 dec)))
    (implies (and (natp itail0)
                  (natp ibytes0))
             (and (n08p byte)
                  (natp itail)
                  (natp ibytes))))
  :rule-classes
  ((:type-prescription
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (natp (nth 0 (x86-64-decode-opcode itail0 ibytes0)))))
   (:type-prescription
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (natp (nth 1 (x86-64-decode-opcode itail0 ibytes0)))))
   (:type-prescription
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (natp (nth 2 (x86-64-decode-opcode itail0 ibytes0)))))
   (:linear
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (<= (nth 0 (x86-64-decode-opcode itail0 ibytes0))
                 #xff)))))

(defthm x86-64-decode-sib-correctness
  (let* ((dec (x86-64-decode-sib itail0 ibytes0 ModR/M))
         (sib (nth 0 dec))
         (itail (nth 1 dec))
         (ibytes (nth 2 dec)))
    (implies (and (natp itail0)
                  (natp ibytes0))
             (and (implies (and (int= (mrm-r/m ModR/M) 4)
                                (not (int= (mrm-mod ModR/M) 3)))
                           (n08p sib))
                  (implies (not (and (int= (mrm-r/m ModR/M) 4)
                                     (not (int= (mrm-mod ModR/M) 3))))
                           (equal sib nil))
                  (natp itail)
                  (natp ibytes))))
  :rule-classes
  ((:type-prescription
    :corollary
    (implies (and (force (natp itail0))
                  (force (natp ibytes0))
                  (equal (mrm-r/m ModR/M) 4)
                  (not (equal (mrm-mod ModR/M) 3)))
             (natp (nth 0 (x86-64-decode-sib itail0 ibytes0 ModR/M)))))
   (:type-prescription
    :corollary
    (implies (and (force (natp itail0))
                  (force (natp ibytes0))
                  (or (not (equal (mrm-r/m ModR/M) 4))
                      (equal (mrm-mod ModR/M) 3)))
             (equal (nth 0 (x86-64-decode-sib itail0 ibytes0 ModR/M))
                    nil)))
   (:type-prescription
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (natp (nth 1 (x86-64-decode-sib itail0 ibytes0 ModR/M)))))
   (:type-prescription
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (natp (nth 2 (x86-64-decode-sib itail0 ibytes0 ModR/M)))))
   (:linear
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (<= (nth 0 (x86-64-decode-sib itail0 ibytes0 ModR/M))
                 #xff)))))

(defthm x86-64-decode-displacement-correctness
  (let* ((dec (x86-64-decode-displacement itail0 ibytes0 ModR/M))
         (displacement (nth 0 dec))
         (itail (nth 1 dec))
         (ibytes (nth 2 dec)))
    (implies (and (natp itail0)
                  (natp ibytes0))
             (and (integerp displacement)
                  (natp itail)
                  (natp ibytes))))
  :rule-classes
  ((:type-prescription
    :corollary
    (implies (and (force (natp itail0))
                  (force (natp ibytes0)))
             (integerp (nth 0 (x86-64-decode-displacement itail0 ibytes0
                                                          ModR/M)))))
   (:type-prescription
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (natp (nth 1 (x86-64-decode-displacement itail0 ibytes0
                                                      ModR/M)))))
   (:type-prescription
    :corollary
    (implies (forced-and (natp itail0)
                         (natp ibytes0))
             (natp (nth 2 (x86-64-decode-displacement itail0 ibytes0
                                                      ModR/M)))))))

(defthm x86-64-decode-operand-nbytes-correctness
  (natp (x86-64-decode-operand-nbytes p3 rex opcode opcode-ext cs-dp
                                      decode-operand-nbytes-ar))
  :rule-classes :type-prescription)

(encapsulate
 ()

 (local (include-book "arithmetic-5/top" :dir :system))

 (local (defthm x86-64-decode-immediate-correctness-lemma
          (implies (and (integerp itail)
                        (integerp operand-nbytes)
                        (<= 0 operand-nbytes)
                        (not (equal operand-nbytes 0)))
                   (< (logand 255 itail)
                      (expt 2 (* 8 operand-nbytes))))))

 (defthm x86-64-decode-immediate-correctness
   (let* ((dec (x86-64-decode-immediate
                itail0 ibytes0 operand-nbytes opcode decode-immediate-ar))
          (immediate (nth 0 dec))
          (ibytes (nth 1 dec)))
     (implies (and (natp ibytes0)
                   (natp operand-nbytes))
              (and (bytesp operand-nbytes immediate)
                   (natp ibytes))))
   :hints (("Goal" :in-theory (enable bytesp)))
   :rule-classes
   ((:rewrite
     :corollary
     (implies (and (natp ibytes0)
                   (natp operand-nbytes))
              (bytesp operand-nbytes
                      (nth 0 (x86-64-decode-immediate
                              itail0 ibytes0 operand-nbytes opcode
                              decode-immediate-ar)))))
    (:type-prescription
     :corollary
     (implies (forced-and (natp ibytes0)
                          (natp operand-nbytes))
              (natp (nth 1 (x86-64-decode-immediate
                            itail0 ibytes0 operand-nbytes opcode
                            decode-immediate-ar)))))))

(local (in-theory (disable opcode-array-p
                           x86-64-decode-displacement
                           x86-64-decode-immediate
                           x86-64-decode-opcode
                           x86-64-decode-operand-nbytes
                           x86-64-decode-prefix
                           x86-64-decode-rex
                           x86-64-decode-sib)))

(defun x86-64-decode (instr 64-bit-modep cs-dp decode-immediate-ar
                            decode-operand-nbytes-ar)
```

```
; Returns (mv erp p1 p2 p3 p4 rex opcode ModR/M sib displacement immediate
; operand-nbytes ModR/M-p ibytes).

;   2. Return instruction fields using mv.  This is where we cause an error if
;      we find we need more than 15 bytes.

;  <= 4: prefix [four groups; each could be nil]
;  <= 1: rex
;  <= 3: opcode
;  <= 1: ModR/M
;  <= 1: SIB [could be nil]
;  <= 4: displacement [signed]
;  <= 8: immediate

; Also returns additional information: operand-nbytes.

  (declare (xargs :guard (and (n120p instr)
                              (opcode-array-p 'decode-immediate
                                              decode-immediate-ar)
                              (opcode-array-p 'decode-operand-nbytes
                                              decode-operand-nbytes-ar))))
  (b* ((ctx 'x86-64-decode)
       ((mv erp p1 p2 p3 p4 itail ibytes)
        (x86-64-decode-prefix instr 0))
       ((when erp)
        (mv (!ms-erp :instr instr)
            0 0 0 0 0 0 0 0 0 0 0 0 0 0))
       ((mv rex itail ibytes)
        (x86-64-decode-rex itail ibytes 64-bit-modep))
       ((mv opcode itail ibytes)
        (x86-64-decode-opcode itail ibytes))
       (ModR/M-p
        (x86-64-decode-ModR/M-p opcode))
       ((mv ModR/M itail ibytes)
        (cond (ModR/M-p
               (get-instruction-byte itail ibytes))
              (t (mv 0 itail ibytes))))
       ((mv sib itail ibytes)
        (x86-64-decode-sib itail ibytes ModR/M))
       ((mv displacement itail ibytes)
        (x86-64-decode-displacement itail ibytes ModR/M))
       (opcode-ext (mrm-reg ModR/M))
       (operand-nbytes
        (x86-64-decode-operand-nbytes p3 rex opcode opcode-ext cs-dp
                                      decode-operand-nbytes-ar))
       ((mv immediate ibytes)
        (x86-64-decode-immediate itail ibytes operand-nbytes opcode
                                 decode-immediate-ar))
       ((when (> ibytes 15))
        (mv (!ms-erp-fresh :instr instr)
            0 0 0 0 0 0 0 0 0 0 0 0 0 0)))
```

```
      (mv nil p1 p2 p3 p4 rex opcode ModR/M sib displacement immediate
          operand-nbytes ModR/M-p ibytes)))

(defun decoded-instruction-p (p1 p2 p3 p4 rex opcode ModR/M sib displacement
                               immediate operand-nbytes ModR/M-p)
  (declare (xargs :guard t))
  (and (n08p? p1)
       (n08p? p2)
       (n08p? p3)
       (n08p? p4)
       (n08p rex)
       (opcodep opcode)
       (n08p ModR/M)
       (if (and (int= (mrm-r/m ModR/M) 4)
                (not (int= (mrm-mod ModR/M) 3)))
           (n08p sib)
         (null sib))
       (integerp displacement)
       (operand-nbytes-p operand-nbytes)
       (bytesp operand-nbytes immediate)
       (equal ModR/M-p (x86-64-decode-ModR/M-p opcode))
       (implies (equal ModR/M-p nil)
                (equal ModR/M 0))))

(defthm decoded-instruction-p-x86-64-decode
  (let ((dec (x86-64-decode instr 64-bit-modep cs-dp decode-immediate-ar
                            decode-operand-nbytes-ar)))
    (implies
     (forced-and (natp instr)
                 (not (nth 0 dec))
                 (equal p1 (nth 1 dec))
                 (equal p2 (nth 2 dec))
                 (equal p3 (nth 3 dec))
                 (equal p4 (nth 4 dec))
                 (equal rex (nth 5 dec))
                 (equal opcode (nth 6 dec))
                 (equal modr/m (nth 7 dec))
                 (equal sib (nth 8 dec))
                 (equal displacement (nth 9 dec))
                 (equal immediate (nth 10 dec))
                 (equal operand-nbytes (nth 11 dec))
                 (equal ModR/M-p (nth 12 dec)))
     (decoded-instruction-p p1 p2 p3 p4 rex opcode modr/m sib displacement
                            immediate operand-nbytes ModR/M-p))))

(defthm decoded-instruction-p-forward-p1
  (implies (and (decoded-instruction-p
                 p1 p2 p3 p4 rex opcode ModR/M sib displacement
                 immediate operand-nbytes ModR/M-p)
                p1)
           (n08p p1))
  :rule-classes :forward-chaining)

(defthm decoded-instruction-p-forward-p2
  (implies (and (decoded-instruction-p
                 p1 p2 p3 p4 rex opcode ModR/M sib displacement
                 immediate operand-nbytes ModR/M-p)
                p2)
           (n08p p2))
  :rule-classes :forward-chaining)

(defthm decoded-instruction-p-forward-p3
  (implies (and (decoded-instruction-p
                 p1 p2 p3 p4 rex opcode ModR/M sib displacement
                 immediate operand-nbytes ModR/M-p)
                p3)
           (n08p p3))
  :rule-classes :forward-chaining)

(defthm decoded-instruction-p-forward-p4
  (implies (and (decoded-instruction-p
                 p1 p2 p3 p4 rex opcode ModR/M sib displacement
                 immediate operand-nbytes ModR/M-p)
                p4)
           (n08p p4))
  :rule-classes :forward-chaining)

(defthm decoded-instruction-p-forward-rex
  (implies (decoded-instruction-p
            p1 p2 p3 p4 rex opcode ModR/M sib displacement
            immediate operand-nbytes ModR/M-p)
           (n08p rex))
  :rule-classes :forward-chaining)

(defthm decoded-instruction-p-forward-opcode
  (implies (decoded-instruction-p
            p1 p2 p3 p4 rex opcode ModR/M sib displacement
            immediate operand-nbytes ModR/M-p)
           (opcodep opcode))
  :rule-classes :forward-chaining)

(defthm decoded-instruction-p-forward-ModR/M
  (implies (decoded-instruction-p
            p1 p2 p3 p4 rex opcode ModR/M sib displacement
            immediate operand-nbytes ModR/M-p)
           (n08p ModR/M))
  :rule-classes :forward-chaining)

(defthm decoded-instruction-p-forward-ModR/M-0
  (implies (and (decoded-instruction-p
                 p1 p2 p3 p4 rex opcode ModR/M sib displacement
                 immediate operand-nbytes ModR/M-p)
                (equal ModR/M-p nil))
           (equal ModR/M 0))
  :rule-classes :forward-chaining)

(defthm decoded-instruction-p-forward-n08p-sib
  (implies (and (decoded-instruction-p
                 p1 p2 p3 p4 rex opcode ModR/M sib displacement
                 immediate operand-nbytes ModR/M-p)
                (or sib
                    (and (equal (logand 7 modr/m) 4)
                         (not (int= (mrm-mod ModR/M) 3)))))
           (n08p sib))
  :rule-classes :forward-chaining)

(defthm decoded-instruction-p-forward-null-sib
  (implies (and (decoded-instruction-p
                 p1 p2 p3 p4 rex opcode ModR/M sib displacement
                 immediate operand-nbytes ModR/M-p)
                (or (not (n08p sib))
                    (not (equal (mrm-r/m modr/m) 4))
                    (int= (mrm-mod ModR/M) 3)))
           (equal sib nil))
  :rule-classes :forward-chaining)

(defthm decoded-instruction-p-forward-displacement
  (implies (decoded-instruction-p
            p1 p2 p3 p4 rex opcode ModR/M sib displacement
            immediate operand-nbytes ModR/M-p)
           (integerp displacement))
  :rule-classes :forward-chaining)

(defthm decoded-instruction-p-forward-immediate
  (implies (decoded-instruction-p
            p1 p2 p3 p4 rex opcode ModR/M sib displacement
            immediate operand-nbytes ModR/M-p)
           (bytesp operand-nbytes immediate))
  :rule-classes :forward-chaining)

(defthm decoded-instruction-p-forward-operand-nbytes
  (implies (decoded-instruction-p
            p1 p2 p3 p4 rex opcode ModR/M sib displacement
            immediate operand-nbytes ModR/M-p)
           (operand-nbytes-p operand-nbytes))
  :rule-classes :forward-chaining)

(defthm decoded-instruction-p-forward-ModR/M-p
  (implies (decoded-instruction-p
            p1 p2 p3 p4 rex opcode ModR/M sib displacement
            immediate operand-nbytes ModR/M-p)
           (equal ModR/M-p (x86-64-decode-ModR/M-p opcode)))
  :rule-classes :forward-chaining)
```

```
(in-theory (disable x86-64-decode decoded-instruction-p))

; Finally, we prove natp-nth-13-x86-64-decode, which says that the ibytes
; returned by the decoder is a natp.  We need this lemma to prove
; x86-64p-x86-64-step (file x86.lisp), which says that our invariant is
; preserved by taking a step.

(deflabel start-natp-nth-13-x86-64-decode)

; Not needed (ACL2 already knows this):
#||
(defthm natp-x86-64-decode-operand-nbytes
  (natp (x86-64-decode-operand-nbytes p3 rex opcode opcode-ext cs-dp
                                      decode-operand-nbytes-ar))
  :hints (("Goal" :in-theory (enable x86-64-decode-operand-nbytes)))
  :rule-classes :type-prescription)
||#

(defthm natp-nth-6-x86-64-decode-prefix-rec
  (implies (force (natp ibytes))
           (natp (nth 6 (x86-64-decode-prefix-rec itail ibytes p1 p2 p3 p4))))
  :hints (("Goal" :in-theory (enable x86-64-decode-prefix-rec)))
  :rule-classes :type-prescription)

(defthm natp-nth-6-x86-64-decode-prefix
  (implies (force (natp ibytes))
           (natp (nth 6 (x86-64-decode-prefix itail ibytes))))
  :hints (("Goal" :in-theory (enable x86-64-decode-prefix)))
  :rule-classes :type-prescription)

(defthm natp-nth-2-x86-64-decode-rex
  (implies (force (natp ibytes))
           (natp (nth 2 (x86-64-decode-rex itail ibytes 64-bit-modep))))
  :hints (("Goal"
           :in-theory
           (e/d (x86-64-decode-rex get-instruction-byte)
                ((force)))))
  :rule-classes :type-prescription)

(defthm natp-nth-2-x86-64-decode-sib
  (implies (force (natp ibytes))
           (natp (nth 2 (x86-64-decode-sib itail ibytes modR/M))))
  :hints (("Goal"
           :in-theory
           (e/d (x86-64-decode-sib get-instruction-byte)
                ((force)))))
  :rule-classes :type-prescription)

(defthm natp-nth-2-x86-64-decode-displacement
  (implies (force (natp ibytes))
           (natp (nth 2 (x86-64-decode-displacement itail ibytes modR/M))))
  :hints (("Goal"
           :in-theory
           (e/d (x86-64-decode-displacement
                 get-instruction-byte
                 get-instruction-dword)
                ((force)))))
  :rule-classes :type-prescription)

(defthm natp-nth-1-x86-64-decode-immediate
  (implies (forced-and (natp ibytes)
                       (natp operand-nbytes))
           (natp (nth 1 (x86-64-decode-immediate
                         itail ibytes operand-nbytes opcode
                         decode-immediate-ar))))
  :hints (("Goal" :in-theory (enable x86-64-decode-immediate)))
  :rule-classes :type-prescription)

(defthm natp-nth-13-x86-64-decode
  (implies (force (natp instr))
           (natp (nth 13 (x86-64-decode
                          instr 64-bit-modep cs-dp decode-immediate-ar
                          decode-operand-nbytes-ar))))
  :hints (("Goal"
           :in-theory
           (union-theories
            '(x86-64-decode mv-nth-is-nth)
            (union-theories
             (current-theory 'ground-zero)
             (set-difference-theories
              (current-theory :here)
              (current-theory 'start-natp-nth-13-x86-64-decode))))))
  :rule-classes :type-prescription)

;   3. Get the effective addresses for memory accesses.

; We will return a 5-tuple for our effective address results:
; (mv erp dst-reg-index dst-mem-addr src-reg-index src-mem-addr).

(defun addr-bits (p4 x86-64)

; See AMD Vol. 1 Table 2-1 p. 18.
; @@ 64-bitp case only

; AMD Vol. 1 says: "Table 2-1 on page 18 shows the effects of using the
; address-size prefix in all operating modes. In 64- bit mode, the default
; address size is 64 bits. The address size can be overridden to 32
; bits. 16-bit addresses are not supported in 64-bit mode. In compatibility and
; legacy modes, the address-size prefix works the same as in the legacy x86
; architecture."
; Intel Vol. 2A Section 2.1.2 p. 2-3: "The address-size override prefix (67H)
; allows programs to switch between 16- and 32-bit addressing. Either size can
; be the default; the prefix selects the non-default size."

; Intel Vol. 1 Sec. 3.6.1 p. 3-26 Table 3-4 reinforces the AMD text above: the
; effective address size in 64-bit mode is always 32 or 64.

; YET: The PUSH instruction in the Intel manual Vol. 2B p. 4-422 (PUSH
; instruction) says that in 64-bit mode, PUSH r/m16 is valid but PUSH r/m32 is
; not encodable.  It also says, for what it's worth: "The address size is used
; only when referencing a source operand in memory."  As documented in
; x86-64-decode-operand-nbytes, the operand size in 64-bit mode for a PUSH
; instruction is 64 bits.  So the "16" must be referring to the address size,
; contradicting mention of "32" in the quotes above!

  (declare (xargs :guard (x86-64p x86-64)
                  :stobjs (x86-64)))
  (declare (ignore x86-64))
  (if p4 32 64))

(defun x86-64-effective-addr-from-sib (addr-bits rex mod sib x86-64)

; Returns a number fitting into the given number of addr-bits.  See Table 2-3,
; Intel Vol. 2A.  The result may be modified with a displacement to form the
; effective address.

;   Quoting Intel Vol. 1 Sec. 3.3.7 ("Address Calculations in 64-Bit Mode")
;   pp. 3-12 to 3-13:

;     All 16-bit and 32-bit address calculations are zero-extended in IA-32e
;     mode to form 64-bit addresses. Address calculations are first truncated
;     to the effective address size of the current mode (64-bit mode or
;     compatibility mode), as overridden by any address-size prefix. The result
;     is then zero-extended to the full 64-bit address width. Because of this,
;     16-bit and 32-bit applications running in compatibility mode can access
;     only the low 4 GBytes of the 64-bit mode effective addresses. Likewise, a
;     32-bit address generated in 64-bit mode can access only the low 4 GBytes
;     of the 64-bit mode effective addresses.

  (declare (xargs :guard (and (posp addr-bits)
                              (n08p rex)
                              (member mod '(0 1 2))
                              (n08p sib)
                              (x86-64p x86-64))
                  :stobjs (x86-64)))
  (b* ((base (b* ((i (n03 sib)))
                 (cond ((and (int= i 5)
                             (int= mod 0))

; See [*] note in Intel Table 2-3.  Note that this case applies regardless of
; REX extension (cf. Intel Table 2-5 p. 2-13).
```

```
                        0)
                       (t (rgfi (reg-index i rex 'b) ; Fig. 2-6, p. 2-12
                                x86-64)))))
       (index (b* ((ix ; use REX prefix (cf. Intel Table 2-5 p. 2-13)
                     (reg-index (n03 (ash sib -3)) rex 'x))) ; Fig. 2-6, p. 2-12
                   (case ix
                     (4 0) ; no index register; "none" in Intel Table 2-3
                     (otherwise (rgfi ix x86-64)))))
       (ss (n02 (ash sib -6)))
       (scaled-index (ash index ss)))
      (n+ addr-bits ; see comment at top of function from Intel manual
          base
          scaled-index)))

(defmacro next-rip (ibytes)
  `(n64+ (rip x86-64) ,ibytes))

(defun x86-64-effective-addr (p4 rex r/m mod sib displacement ibytes x86-64)

; Returns the effective address (a number).  See Intel Table 2-2, p. 2-7.

; Note that this computation is done before using paging and (for other than
; 64-bit mode) segmentation.  AMD Vol. 1 p. 15 (Sec. 2.2.3) says: "Programs
; provide effective addresses to the hardware prior to segmentation and paging
; translations."  See also AMD Vol. 1 p. 12 Fig. 2-3, and also Fig. 2-4 on the
; next page, which use "effective address" to denote input to the segmentation
; calculation.  Our "effective address" functions use the ModR/M and SIB bytes,
; as described on p. 16.

  (declare (xargs :guard (and (n08p rex)
                              (n03p r/m)
                              (member mod '(0 1 2))
                              (if (int= r/m 4)
                                  (n08p sib)
                                (null sib))
                              (integerp displacement)
                              (ibytesp ibytes)
                              (x86-64p x86-64))
                  :stobjs (x86-64)))
  (b* ((addr-bits (addr-bits p4 x86-64))
       (dst-base
        (case r/m
          (4 ; regardless of REX extension (cf. Intel Table 2-5 p. 2-13)
           (x86-64-effective-addr-from-sib addr-bits rex mod sib x86-64))
          (otherwise
           (cond ((and (int= mod 0)
                       (int= r/m 5))

; regardless of REX extension (cf. Intel Table 2-5 p. 2-13)
```

```
                 (cond ((64-bit-modep x86-64)

; RIP-relative addressing; AMD Vol. 3 Sec. 1.7 pp. 24-25, and
; Intel Vol. 2A Sec. 2.2.1.6.

; Note that even in this case, we still do address calculations relative to 32
; bits when appropriate.  That seems pretty clear from reading the manuals, but
; is confirmed by this, from
; http://x86asm.net/articles/x86-64-tour-of-intel-manuals/#Default-Operand-and-Address-Size:

;   .... Another
;   (not so obvious) aspect of RIP-relative addressing is fact, that it is
;   (just like any other addressing) controlled by address-size override prefix
;   67. With this prefix, it is possible to address relative to EIP:

;   67 8B 05 10 00 00 00   MOV EAX, [EIP+10h]

;   This is not described anywhere in manuals directly, nor called EIP-relative
;   addressing.

                        (next-rip ibytes))
                       (t 0)))
                (t (rgfi (reg-index r/m rex 'b) ; Intel Fig. 2-4, p. 2-11
                        x86-64)))))))
      (n+ addr-bits dst-base displacement)))

(defthm n64p-x86-64-effective-addr
  (n64p (x86-64-effective-addr p4 rex r/m mod sib displacement ibytes x86-64))
  :hints

; We disable the :executable-counterpart of force in order to instruct the
; system to disable forcing (see :DOC disable-forcing).  Otherwise rules like
; NATP-RGFI can cause an impossible forcing round because we don't know
; (x86-64p x86-64).

  (("Goal" :in-theory (disable (force))))
  :rule-classes
  ((:type-prescription
    :corollary
    (natp (x86-64-effective-addr p4 rex r/m mod sib displacement ibytes x86-64)))
   (:linear
    :corollary
    (let ((addr (x86-64-effective-addr p4 rex r/m mod sib displacement ibytes
                                       x86-64)))
      (< addr *2^64*)))))

(in-theory (disable x86-64-effective-addr))

(defun x86-64-effective-addr-and-regs (p4 rex opcode ModR/M ModR/M-p sib
                                          displacement ibytes x86-64)
```

```
; Returns (mv r/m-index r/m-addr reg-index), where each field is either nil or
; a number.

  (declare (xargs :guard (and (n08p rex)
                              (opcodep opcode)
                              (n08p ModR/M)
                              (let ((r/m (mrm-r/m ModR/M))
                                    (mod (mrm-mod ModR/M)))
                                (if (and (int= r/m 4)
                                         (not (int= mod 3)))
                                    (n08p sib)
                                  (null sib)))
                              (integerp displacement)
                              (ibytesp ibytes)
                              (x86-64p x86-64))
                  :stobjs (x86-64)))
  (cond
   (ModR/M-p
    (b* ((r/m (mrm-r/m ModR/M))
         (regx ; warning: use 'b when Reg is from Opcode
          (reg-index (mrm-reg ModR/M) rex 'r))
         (mod (mrm-mod ModR/M)))
        (case mod
          (3 (mv (reg-index r/m rex 'b) nil regx)) ; Intel Fig 2-5, p. 2-11
          (otherwise
           (mv nil
               (x86-64-effective-addr p4 rex r/m mod sib displacement ibytes
                                      x86-64)
               regx)))))
   ((member opcode '(            ; always target AL, AX, EAX, or RAX
                     #x04 #x05   ; ADD
                     #x14 #x15)) ; ADC
    (mv nil nil 0))
   (t ; +r_ case, Intel Vol. 2, Sec. 3.1.1.1 p. 3-2 and Fig 2-7 p. 2-12
    (mv nil
        nil
        (reg-index (n03 opcode) rex 'b)))))

(defthm x86-64-effective-addr-types
  (mv-let (dst-reg-index dst-addr src-reg-index src-addr)
          (x86-64-effective-addr-and-regs p4 rex opcode ModR/M ModR/M-p sib
                                          displacement ibytes x86-64)
          (and (n04p? dst-reg-index)
               (n64p? dst-addr)
               (n04p? src-reg-index)
               (n64p? src-addr)))
  :rule-classes

; We don't need this lemma, because we have decided to leave
; x86-64-effective-addr-and-regs enabled, disabling x86-64-effective-addr
; instead, which is enough to speed up the guard proof for x86-64-step-mov
; without.  But we leave this lemma around anyhow, merely commenting out its
; :rule-classes.

  nil

#||
  ((:type-prescription
    :corollary
    (or (equal (nth 0 (x86-64-effective-addr-and-regs
                       p4 rex ModR/M sib displacement ibytes x86-64))
               nil)
        (natp (nth 0 (x86-64-effective-addr-and-regs
                      p4 rex ModR/M sib displacement ibytes x86-64)))))
   (:linear
    :corollary
    (let ((dst-reg-index (nth 0 (x86-64-effective-addr-and-regs
                                 p4 rex ModR/M sib displacement ibytes x86-64))))
      (implies dst-reg-index
               (<= dst-reg-index 15))))
   (:type-prescription
    :corollary
    (or (equal (nth 1 (x86-64-effective-addr-and-regs
                       p4 rex ModR/M sib displacement ibytes x86-64)) nil)
        (natp (nth 1 (x86-64-effective-addr-and-regs
                      p4 rex ModR/M sib displacement ibytes x86-64)))))
   (:linear
    :corollary
    (let ((dst-addr (nth 1 (x86-64-effective-addr-and-regs
                            p4 rex ModR/M sib displacement ibytes x86-64))))
      (implies dst-addr
               (< dst-addr *2^64*))))
   (:type-prescription
    :corollary
    (or (equal (nth 2 (x86-64-effective-addr-and-regs
                       p4 rex ModR/M sib displacement ibytes x86-64)) nil)
        (natp (nth 2 (x86-64-effective-addr-and-regs
                      p4 rex ModR/M sib displacement ibytes x86-64)))))
   (:linear
    :corollary
    (let ((src-reg-index (nth 2 (x86-64-effective-addr-and-regs
                                 p4 rex ModR/M sib displacement ibytes x86-64))))
      (implies src-reg-index
               (<= src-reg-index 15))))
   (:type-prescription
    :corollary
    (or (equal (nth 3 (x86-64-effective-addr-and-regs
                       p4 rex ModR/M sib displacement ibytes x86-64)) nil)
        (natp (nth 3 (x86-64-effective-addr-and-regs
                      p4 rex ModR/M sib displacement ibytes x86-64)))))
   (:linear
    :corollary
    (let ((src-addr (nth 3 (x86-64-effective-addr-and-regs
                            p4 rex ModR/M sib displacement ibytes x86-64))))
      (implies src-addr
               (< src-addr *2^64*)))))
||#
  )

; We leave x86-64-effective-addr-and-regs enabled because its various cases are
; needed in the guard proof for the MOV instruction function, x86-64-step-mov.
; (in-theory (disable x86-64-effective-addr-and-regs))

;   4. Check read and write permissions.

(defun x86-64-permission-p (addr nbytes rw x86-64)

; Returns a Boolean indicating if we can read or write nbytes starting at addr
; (reading and/or writing, according to rw).  Note that at most one of dst-addr
; and src-addr will be non-nil.  Later we'll have a more elaborate check having
; to do with more genuine permissions checking.

  (declare (xargs :guard (and (n64p addr)
                              (operand-nbytes-p nbytes)
                              (member-eq rw '(r w rw nil))
                              (x86-64p x86-64))
                  :stobjs (x86-64)))
  (declare (ignore rw x86-64))
  (<= addr (- *2^48* nbytes)))

;   5. Branch on opcode and call the individual instruction update functions.

; We're almost ready to handle individual instructions.  First we define some
; utilities.

; See Intel Secs. 3.2 and 4.2.

(defmacro !!rip (form)

; Advance the rip by ibytes.  Note that ibytes, ctx, and x86-64 should be bound
; when calling this macro.

  `(b* ((next-rip (next-rip ibytes))
        (x86-64 ,form)
        ((when (>= next-rip *2^48*))
         (!!ms-fresh :ibytes ibytes)))
       (!rip next-rip x86-64)))

(defthm x86-64p-!ms

; First needed for x86-64p-x86-64-step-mov.

  (implies (force (x86-64p x86-64))
          (x86-64p (update-nth *ms* v x86-64)))
  :hints (("Goal" :in-theory (enable x86-64p
                                     x86-64p-pre
                                     mem-array-next-addr))))

(defun src-dst-fields-1 (r/m-index r/m-addr reg-index opcode opcode-ext
                                   src-dst-fields-ar)

; Returns (mv dst-index dst-addr src-index src-addr rw), where rw is a suitable
; value to go with a non-nil r/m-addr in a call of x86-64-permission-p.  We
; pass in some combination of an effective address and register indices (where
; each parameter could be nil, and not more than one of r/m-index and r/m-addr
; is non-nil).  Thus, if r/m-addr is returned as a source and/or destination,
; then the returned rw is meaningful; otherwise the returned rw is a
; don't-care.

; For ops such as ADD, where a register or memory address can serve as both a
; source and a destination, we treat such as a destination; the other operand,
; if any, is the source.  Similarly, for ops such as CMP, where there is no
; destination, we treat one of the ops as a "destination".

; Technical note: At one time we tried disabling array1p, but it needed to be
; enabled for the guard conjecture.

  (declare (xargs :guard (and (n04p? r/m-index)
                              (n64p? r/m-addr)
                              (n04p? reg-index)
                              (opcodep opcode)
                              (n03p opcode-ext)
                              (opcode-array-p 'src-dst-fields
                                              src-dst-fields-ar))
                  :guard-hints (("Goal" :in-theory (enable opcode-array-p)))))
  (case (opcode-array-element 'src-dst-fields src-dst-fields-ar opcode
                              opcode-ext)
;      (mv dst-index dst-addr src-index src-addr rw)
    (0 (mv r/m-index r/m-addr reg-index      nil 'w))
    (1 (mv reg-index      nil r/m-index r/m-addr 'r))
    (2 (mv r/m-index r/m-addr       nil      nil 'w))
    (3 (mv reg-index      nil       nil      nil nil))
    (4 (mv       nil      nil reg-index      nil nil))
    (5 (mv r/m-index r/m-addr       nil      nil 'rw))
    (6 (mv r/m-index r/m-addr reg-index      nil 'rw))
    (7 (mv reg-index      nil r/m-index r/m-addr nil))
    (otherwise ; @@ stub or don't-care (no operands, e.g., #xf4 and #6a)
     (mv   r/m-index r/m-addr reg-index      nil 'rw))))

(defun src-dst-fields (p4 rex opcode ModR/M ModR/M-p sib displacement ibytes
                       src-dst-fields-ar operand-nbytes x86-64)
  (declare (xargs :guard (and (n08p rex)
                              (opcodep opcode)
                              (n08p ModR/M)
```

```
                            (let ((r/m (mrm-r/m ModR/M))
                                  (mod (mrm-mod ModR/M)))
                              (if (and (int= r/m 4)
                                       (not (int= mod 3)))
                                  (n08p sib)
                                (null sib)))
                            (integerp displacement)
                            (ibytesp ibytes)
                            (opcode-array-p 'src-dst-fields
                                            src-dst-fields-ar)
                            (operand-nbytes-p operand-nbytes)
                            (x86-64p x86-64))
                :stobjs (x86-64)))
(b* ((opcode-ext (mrm-reg ModR/M))
     ((mv r/m-index r/m-addr reg-index)
      (x86-64-effective-addr-and-regs p4 rex opcode ModR/M ModR/M-p
                                      sib displacement ibytes
                                      x86-64))
     ((mv dst-index dst-addr src-index src-addr rw)
      (src-dst-fields-1 r/m-index r/m-addr reg-index opcode opcode-ext
                        src-dst-fields-ar))
     (permission-p (or (null r/m-addr)
                       (x86-64-permission-p
                        r/m-addr operand-nbytes rw x86-64))))
    (mv (not permission-p) dst-index dst-addr src-index src-addr)))
```

16 ========== File x86-64/x86.lisp ==========

```
; See file TODO for things to do.

; Follow these steps to add a new instruction step function:

; (1) Update *op-alist*, *decode-immediate-lst*, *decode-operand-nbytes-lst*,
;     and *src-dst-fields-lst* below.  Set a table entry to - for "don't
;     care" (for each of the three *xxx-lst* tables).

; (2) If the opcode accesses registers in an unusual way (i.e. using neither
;     the ModR/M nor the low 3 bits of the opcode, as in ADD #x04 and #x05),
;     modify x86-64-effective-addr-and-regs accordingly.

; (3) Add a defstep form; see e.g. (defstep mov ...).

(in-package "ACL2")

(include-book "x86-general")

; Constant arrays for instructions

(defconst *op-alist*
  '(
    (ADC/ADD/AND/CMP/OR/SBB/SUB/TEST/XOR
     #x00 #x01 #x02 #x03   ; ADD
     #x04 #x05             ; ADD
     #x08 #x09 #x0a #x0b   ; OR
     #x0c #x0d             ; OR
     #x10 #x11 #x12 #x13   ; ADC
     #x14 #x15             ; ADC
     #x18 #x19 #x1a #x1b   ; SBB
     #x1c #x1d             ; SBB
     #x20 #x21 #x22 #x23   ; AND
     #x24 #x25             ; AND
     #x28 #x29 #x2a #x2b   ; SUB
     #x2c #x2d             ; SUB
     #x30 #x31 #x32 #x33   ; XOR
     #x34 #x35             ; XOR
     #x38 #x39 #x3a #x3b   ; CMP
     #x3c #x3d             ; CMP
     #x80 #x81 #x83        ; all
     #x84 #x85             ; TEST
     #xa8 #xa9             ; TEST
     (#xf6 . 0) (#xf7 . 0) ; TEST
     )
    (CALL #xe8)
    (HLT #xf4)
    (INC ; (#xfe . 0) ; but not #x40+r in 64-bit mode
     (#xff . 0))
    (JCC #x74  ; JE rel8  (jump if ZF=1)
         #x75  ; JNE rel8 (jump if ZF=0)
         #x7e  ; JLE rel8 (jump if ZF=1 or SF!=OF)
         #x7f) ; JG rel8  (jump if ZF=0 and SF=OF)
    (JMP #xe9 #xeb)
    (LEA #x8d)
    (LEAVE #xc9)
    (MOV #x88 #x89 #x8a #x8b
         #xb0 #xb1 #xb2 #xb3 #xb4 #xb5 #xb6 #xb7
         #xb8 #xb9 #xba #xbb #xbc #xbd #xbe #xbf
         #xc6 #xc7)
    (NOP #x90)
    (POP #x58 #x59 #x5a #x5b #x5c #x5d #x5e #x5f
         #x8f)
    (PUSH #x50 #x51 #x52 #x53 #x54 #x55 #x56 #x57
          #x68
          #x6a
          (#xff . 6))
    (RET #xc3)))

(defmacro defopcodes ()

; This macro generates predicates such as PUSH$OPCODEP and a suitable defthm
; and in-theory for that predicate.  (See defopcodes-fn.)

; Warning: the same opcode can be recognized by more than one FOO$OPCODEP
; function.  (But we see no problem with that, for our current uses.)

  (defopcodes-fn *op-alist* nil))

(defopcodes)

; Wart: f6 and f7 in *decode-immediate-lst*, and perhaps others, may need the
; 3-bit "opcode-ext" extension, e.g., f6/0 and f7/0 (for TEST).

(defconst *decode-immediate-lst*

; This table tells us the case to use in x86-64-decode-immediate.  0 and 1
; indicate 0 and 1 bytes, respectively.  See x86-64-decode-immediate for any
; other cases.

 '(
   #|       0 1 2 3 4 5 6 7 8 9 a b c d e f        |#
   #|       -------------------------------        |#
   #| 00 |# 0 0 0 0 1 3 x x 0 0 0 0 1 3 x x  #| 00 |#
   #| 10 |# 0 0 0 0 1 3 x x 0 0 0 0 1 3 x x  #| 10 |#
   #| 20 |# 0 0 0 0 1 3 x x 0 0 0 0 1 3 x x  #| 20 |#
   #| 30 |# 0 0 0 0 1 3 x x 0 0 0 0 1 3 x x  #| 30 |#
   #| 40 |# x x x x x x x x x x x x x x x x  #| 40 |#
   #| 50 |# 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  #| 50 |#
   #| 60 |# x x x x x x x x 2 x 1 x x x x x  #| 60 |#
   #| 70 |# x x x x 1 1 x x x x x x x x 1 1  #| 70 |#
   #| 80 |# 1 3 x 1 0 0 x x 0 0 0 0 x 0 x 0  #| 80 |#
   #| 90 |# 0 x x x x x x x x x x x x x x x  #| 90 |#
   #| a0 |# x x x x x x x x 1 3 x x x x x x  #| a0 |#
   #| b0 |# 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2  #| b0 |#
   #| c0 |# x x x 0 x x 1 3 x 0 x x x x x x  #| c0 |#
   #| d0 |# x x x x x x x x x x x x x x x x  #| d0 |#
   #| e0 |# x x x x x x x x 2 3 x 1 x x x x  #| e0 |#
   #| f0 |# x x x x 0 x 1 3 x x x x x x x 0  #| f0 |#
   #|       -------------------------------        |#
   #|       0 1 2 3 4 5 6 7 8 9 a b c d e f        |#
   ))

(defconst *decode-immediate-ar*
  (list-to-array 'decode-immediate *decode-immediate-lst*))

(defconst *decode-operand-nbytes-lst*

; This table tells us the case to use in x86-64-decode-operand-nbytes; see that
; function for the meanings of the values in this table.  WARNING: as explained
; in that function (and as required in the definition of
; x86-64-decode-immediate), do not use 0 for the operand width if there is an
; immediate.

 '(
   #|       0 1 2 3 4 5 6 7 8 9 a b c d e f        |#
   #|       -------------------------------        |#
   #| 00 |# 1 2 1 2 1 2 x x 1 2 1 2 1 2 x x  #| 00 |#
   #| 10 |# 1 2 1 2 1 2 x x 1 2 1 2 1 2 x x  #| 10 |#
   #| 20 |# 1 2 1 2 1 2 x x 1 2 1 2 1 2 x x  #| 20 |#
   #| 30 |# 1 2 1 2 1 2 x x 1 2 1 2 1 2 x x  #| 30 |#
   #| 40 |# x x x x x x x x x x x x x x x x  #| 40 |#
   #| 50 |# 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3  #| 50 |#
   #| 60 |# x x x x x x x x 3 x 3 x x x x x  #| 60 |#
   #| 70 |# x x x x 1 1 x x x x x x x x 1 1  #| 70 |#
   #| 80 |# 1 2 x 2 1 2 x x 1 2 1 2 x 2 x 3  #| 80 |#
   #| 90 |# 0 x x x x x x x x x x x x x x x  #| 90 |#
   #| a0 |# x x x x x x x x 1 2 x x x x x x  #| a0 |#
   #| b0 |# 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2  #| b0 |#
   #| c0 |# x x x 0 x x 1 2 x 3 x x x x x x  #| c0 |#
   #| d0 |# x x x x x x x x x x x x x x x x  #| d0 |#
   #| e0 |# x x x x x x x x 4 4 x 1 x x x x  #| e0 |#
   #| f0 |# x x x x 0 x                      #|    |#
   #|-f6-|# (1 x x x x x x x)                #|-f6-|#
   #|-f7-|# (2 x x x x x x x)                #|-f7-|#
   #|    |#                 x x x x x x x x  #|    |#
   #|-ff-|# (2 x x x x x x 3 x)              #|-ff-|#
   #|       -------------------------------        |#
   #|       0 1 2 3 4 5 6 7 8 9 a b c d e f        |#
   )
)
```

```
(defconst *decode-operand-nbytes-ar*
  (list-to-array 'decode-operand-nbytes *decode-operand-nbytes-lst*))

(defconst *src-dst-fields-lst*

; This table tells us the case to use in function src-dst-fields-1.

 '(
   #|       0 1 2 3 4 5 6 7 8 9 a b c d e f        |#
   #|       -------------------------------        |#
   #| 00 |# 6 6 1 1 3 3 x x 6 6 1 1 3 3 x x  #| 00 |#
   #| 10 |# 6 6 1 1 3 3 x x 6 6 1 1 3 3 x x  #| 10 |#
   #| 20 |# 6 6 1 1 3 3 x x 6 6 1 1 3 3 x x  #| 20 |#
   #| 30 |# 6 6 1 1 3 3 x x 6 6 1 1 3 3 x x  #| 30 |#
   #| 40 |# x x x x x x x x x x x x x x x x  #| 40 |#
   #| 50 |# 4 4 4 4 4 4 4 4 3 3 3 3 3 3 3 3  #| 50 |#
   #| 60 |# x x x x x x x x - x - x x x x x  #| 60 |#
   #| 70 |# x x x x - - x x x x x x x x - -  #| 70 |#
   #| 80 |# 5 5 x 5 6 6 x x 0 0 1 1 x 7 x 2  #| 80 |#
   #| 90 |# x x x x x x x x x x x x x x x x  #| 90 |#
   #| a0 |# x x x x x x x x 3 3 x x x x x x  #| a0 |#
   #| b0 |# 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3  #| b0 |#
   #| c0 |# x x x - x x 2 2 x - x x x x x x  #| c0 |#
   #| d0 |# x x x x x x x x x x x x x x x x  #| d0 |#
   #| e0 |# x x x x x x x x - - x - x x x x  #| e0 |#
   #| f0 |# x x x x - x                      #|    |#
   #|-f6-|# (5 x x x x x x x)                #|-f6-|#
   #|-f7-|# (5 x x x x x x x)                #|-f7-|#
   #|    |#                 x x x x x x x    #|    |#
   #|-ff-|# (5 x x x x x 1 x)                #|-ff-|#
   #|       -------------------------------        |#
   #|       0 1 2 3 4 5 6 7 8 9 a b c d e f        |#
   ))

(defconst *src-dst-fields-ar*
  (list-to-array 'src-dst-fields *src-dst-fields-lst*))

(defun x86-64-step-unimplemented (opcode x86-64)
  (declare (xargs :guard (x86-64p x86-64)
                  :stobjs (x86-64)))
  (b* ((ctx 'x86-64-step-unimplemented))
      (!!ms-fresh :opcode opcode)))

(defthm x86-64p-x86-64-step-unimplemented
  (implies (and (x86-64p x86-64)
                (not (nth *ms* x86-64)))
           (x86-64p (x86-64-step-unimplemented opcode x86-64)))
  :hints (("Goal"
           :in-theory (enable x86-64p x86-64p-pre mem-array-next-addr))))

(in-theory (disable x86-64-step-unimplemented))
```

```
(defmacro defstep (op-name &rest rst)

; Defstep lays down appropriate definitions and theorems for the given op-name,
; for example MOV, which must be bound in *op-alist*.  The definition of the
; step function includes guard and stobj information common to all such step
; functions, including (in essence) the forms (natp ibytes) and (x86-64p
; x86-64).

  (cond
   ((not (assoc-eq op-name *op-alist*))
    (er hard 'defstep
        "It is only legal to call defstep after the op name has been put into ~
         *op-alist*, but this is not the case for: ~x0."
        op-name))
   (t (let* ((args `(,@*decoded-fields* ibytes x86-64))
             (step-name (x86-64-step-name op-name))
             (opcode-recognizer (packn (list op-name "$OPCODEP")))
             (thm-name (packn (list 'x86-64p- step-name))))
        `(progn (defun ,step-name ,args
                  (declare (ignorable ,@*decoded-fields* ibytes)
                           (type (integer 0 *) ibytes)
                           (xargs :stobjs x86-64
                                  :guard
                                  (and (decoded-instruction-p
                                         ,@*decoded-fields*)
                                       (,opcode-recognizer opcode)
                                       (x86-64p x86-64))))
                  ,@(butlast rst 1)
                  (let ((ctx ',step-name))
                    (declare (ignorable ctx))
                    ,(car (last rst))))
                (defthm ,thm-name
                  (implies
                   (forced-and (decoded-instruction-p ,@*decoded-fields*)
                               (,opcode-recognizer opcode)
                               (natp ibytes)
                               (x86-64p x86-64))
                   (x86-64p (,step-name ,@args))))
                (in-theory (disable ,step-name)))))))

; Wart: Maybe not needed if we prove suitable stuff?
(in-theory (enable opcode-array-element))

; Start instructions

(defun arith-result (opcode opcode-ext src dst x86-64)
  (declare (xargs :guard (and (opcodep opcode)
                              (natp src)
                              (natp dst)
                              (x86-64p x86-64))
```

```
                 :stobjs (x86-64)))
  (case opcode
    ((#x00 #x01 #x02 #x03 ; ADD
           #x04 #x05)     ; ADD
     (+ src dst))
    ((#x08 #x09 #x0a #x0b ; OR
           #x0c #x0d)     ; OR
     (logior src dst))
    ((#x10 #x11 #x12 #x13 ; ADC
           #x14 #x15)     ; ADC
     (+ src dst (x86-eflags :cf (flg x86-64))))
    ((#x18 #x19 #x1a #x1b ; SBB
           #x1c #x1d)     ; SBB
     (- dst
        (+ src (x86-eflags :cf (flg x86-64)))))
    ((#x20 #x21 #x22 #x23       ; AND
           #x24 #x25            ; AND
           #xf6 #xf7            ; TEST (wart: only for opcode-ext = 0)
           #xa8 #xa9 #x84 #x85) ; TEST
     (logand src dst))
    ((#x28 #x29 #x2a #x2b #x38 #x39 #x3a #x3b ; SUB, CMP
           #x2c #x2d #x3c #x3d)               ; SUB, CMP
     (- dst src))
    ((#x30 #x31 #x32 #x33 ; XOR
           #x34 #x35)     ; XOR
     (logxor src dst))
    ((#x80 #x81 #x83) ; see Intel Vol. 2 Sec. A.4.2 Table A-6 p. A-21
     (case opcode-ext
       (0 (+ src dst))                                   ; ADD
       (1 (logior src dst))                              ; OR
       (2 (+ src dst (x86-eflags :cf (flg x86-64))))     ; ADC
       (3 (- dst (+ src (x86-eflags :cf (flg x86-64))))) ; SBB
       (4 (logand src dst))                              ; AND
       ((5 7) (- dst src))                               ; SUB, CMP
       (otherwise                                        ; 6, XOR
        (logxor src dst))))
    (otherwise ; impossible case
     0)))

(defthm integerp-arith-result
  (implies (forced-and (natp src1)
                       (natp src2)
                       (x86-64p x86-64))
           (integerp (arith-result opcode opcode-ext src1 src2 x86-64)))
  :rule-classes :type-prescription)

(in-theory (disable arith-result))

(defun eflags-result (raw-result result operand-nbits src1 src2 opcode
                                 opcode-ext x86-64)
```

```
; Intel Vol. 1 Sec. 3.4.3.1 p. 3-21 says: "The status flags (bits 0, 2, 4, 6,
; 7, and 11) of the EFLAGS register indicate the results of arithmetic
; instructions, such as the ADD, SUB, MUL, and DIV instructions."  Those bits
; are shown as (resp.) CF, PF, AF, ZF, SF, and OF.  Result is obtained by
; truncating raw-result (which is produced without truncation) to
; operand-nbits.

  (declare (xargs :guard (and (integerp raw-result)
                              (integerp result)
                              (natp operand-nbits)
                              (natp src1)
                              (natp src2)
                              (opcodep opcode)
                              (x86-64p x86-64))
                  :stobjs (x86-64)))
  (let* ((operand-nbits

; Wart: We know already that operand-nbits >= 8.  But the way we are passing
; guards around, that isn't necessarily the case here, or in its caller.

          (max 1 operand-nbits))
         (cf (<= (ash 1 operand-nbits) raw-result)))
    (!arith-flags
     (flg x86-64)
     :cf (if (and (int= opcode #xff)
                  (eql opcode-ext 0)) ; INC
             (x86-eflags :cf (flg x86-64))
           cf)
     :pf (evenp (logcount (logand #xff raw-result)))
; Wart: :af is undefined for some ops
     :af (< #xf (+ (logand #xf src1) (logand #xf src2)))
     :zf (int= result 0)
     :sf (logbitp (1- operand-nbits) raw-result)
     :of
     (case opcode
       ((
         #x08 #x09 #x0a #x0b #x0c #x0d ; OR
         #x20 #x21 #x22 #x23 #x24 #x25 ; AND
         #x30 #x31 #x32 #x33 #x34 #x35 ; XOR
         #xf6 #xf7                     ; TEST (wart: only for opcode-ext = 0)
         #xa8 #xa9 #x84 #x85)          ; TEST
        nil)
       (otherwise
        (cond ((and (member opcode '(#x80 #x81 #x83))
                    (member opcode-ext '(1 4 6))) ; OR, AND, XOR
               nil)
              (t

; Wart: Is this correct for SUB and SBB?  Also double-check for the others (got
; this from Wikipedia).
```

```
              (xor cf
                   (b* ((bound (ash 1 (1- operand-nbits)))
                        (mask (1- bound)))
                       (<= bound
                           (+ (logand mask src1) (logand mask src2)))))))))))

(in-theory (disable flg-val logbitp))

(defthm n64p-eflags-result-type-prescription
  (natp (eflags-result raw-result result operand-nbits src1 src2
                       opcode opcode-ext x86-64))
  :rule-classes :type-prescription)

(defthm n64p-eflags-result-type-linear
  (implies (forced-and (x86-64p x86-64))
           (<= (eflags-result raw-result result operand-nbits src1 src2
                              opcode opcode-ext x86-64)
               *2^64-1*))
  :hints (("Goal"
           :in-theory
           (enable
            (:linear n64p-logior-n64p-less-than-2^64))))
  :rule-classes :linear)

(in-theory (disable eflags-result))

(defthm reg-indexp-0
  (reg-indexp 0 rex)
  :hints (("Goal" :in-theory (enable reg-indexp))))

(defthm nth-quotep-2
  (implies (and (syntaxp (quotep x))
                (consp x))
           (equal (nth n x)
                  (if (zp n)
                      (car x)
                    (nth (1- n) (cdr x)))))
  :hints (("Goal" :in-theory (enable nth))))

(defstep adc/add/and/cmp/or/sbb/sub/test/xor
  (b* (((mv flg dst-index dst-addr src-index src-addr)
        (src-dst-fields p4 rex opcode ModR/M ModR/M-p sib displacement ibytes
                        *src-dst-fields-ar* operand-nbytes x86-64))
       ((when flg)
        (!!ms-fresh :permission nil))
       (opcode-ext (mrm-reg ModR/M))
       (operand-nbits (ash operand-nbytes 3))
       (src
        (cond (src-index ; read from register
               (rgfi-size operand-nbytes src-index rex x86-64))
              (src-addr ; read from memory
               (rm-size operand-nbytes src-addr x86-64))
              ((int= opcode #x83)

; See e.g. Intel Vol. 2 ADD instruction (similarly for others with different
; 3-bit extension from the #x83 family).

               (sign-extend immediate 8 operand-nbits))
              ((int= operand-nbytes 8) ; sign-extend from 32 to 64 bits

; Note: Opcode #x80 is shown in Intel Vol. 2 as having operand-nbytes = 8 when
; there is a rex prefix.  But it doesn't say "REX.W", so this is suspicious.
; Indeed, the AMD manual Vol. 3, p. 79 shows only an 8-bit operand for #x80.

               (sign-extend immediate 32 64))
              (t immediate)))
        (dst
         (cond (dst-index ; read/write to register
                (rgfi-size operand-nbytes dst-index rex x86-64))
               (dst-addr ; read/write to memory
                (rm-size operand-nbytes dst-addr x86-64))
               (t ; impossible case, as proved by guard verification
                (er hard ctx
                    "dst-addr and dst-index are both nil."))))
        (raw-result (arith-result opcode opcode-ext src dst x86-64))
        (max-result (1- (ash 1 operand-nbits)))
        (result (logand raw-result max-result))
        (eflags (eflags-result raw-result result operand-nbits src dst opcode
                               opcode-ext x86-64))
        (x86-64 (!flg eflags x86-64)))
       (!!rip
        (cond ((or (member opcode '(#x38 #x39 #x3a #x3b #x3c #x3d)) ; CMP
                   (and (member opcode '(#x80 #x81 #x83))
                        (int= opcode-ext 7))             ; CMP
                   (member opcode '(#xa8 #xa9 #x84 #x85)) ; TEST
                   (and (member opcode '(#xf6 #xf7))
                        (int= opcode-ext 0))) ; TEST
              x86-64)
             (dst-index ; write to register
              (!rgfi-size operand-nbytes dst-index result rex x86-64))
             (dst-addr ; write to memory
              (wm-size operand-nbytes dst-addr result x86-64))
             (t ; impossible case, as proved by guard verification
              (prog2$ (er hard ctx
                          "dst-addr and dst-index are both nil.")
                      x86-64))))))

(defstep call
  (b* ((next-rip (next-rip ibytes))
       ((when (> next-rip *2^48-16*)) ; bad return rip; let's error now
        (!!ms-fresh :next-rip next-rip))
       (displacement (sign-extend immediate 32 64))
       (call-rip (n64+ next-rip displacement))
       ((when (> call-rip *2^48-16*)) ; bad call rip; let's error now
        (!!ms-fresh :call-rip call-rip))
       (rsp (rgfi *mr-rsp* x86-64))
       ((when (> rsp *2^48*)) ; see PUSH code for explanation
        (!!ms-fresh :rsp rsp))
       (new-rsp (- rsp 8))
       ((when (< new-rsp 0))
        (!!ms-fresh :rsp rsp :new-rsp new-rsp))
; start push
       (x86-64 (wm64 new-rsp next-rip x86-64))
       (x86-64 (!rgfi *mr-rsp* new-rsp x86-64))
; end push
       (x86-64 (!rip call-rip x86-64)))
      x86-64))

(defstep hlt
  (!!ms-fresh :hlt :legal-halt))

(defstep inc
  (b* (((mv flg dst-index dst-addr src-index src-addr)
        (src-dst-fields p4 rex opcode ModR/M ModR/M-p sib displacement ibytes
                        *src-dst-fields-ar* operand-nbytes x86-64))
       ((when flg)
        (!!ms-fresh :permission nil))
       ((when (or src-index src-addr))
        (!!ms-fresh :unexpected-case t :src-index src-index :src-addr src-addr))
       (operand-nbits (ash operand-nbytes 3))
       (dst
        (cond (dst-index ; write to register
               (rgfi-size operand-nbytes dst-index rex x86-64))
              (dst-addr ; write to memory
               (rm-size operand-nbytes dst-addr x86-64))
              (t ; impossible case, as proved by guard verification
               (er hard ctx
                   "dst-addr and dst-index are both nil."))))
       (raw-result (1+ dst))
       (max-result (1- (ash 1 operand-nbits)))
       (result (logand raw-result max-result))
       (opcode-ext (mrm-reg ModR/M))
       (eflags (eflags-result raw-result result operand-nbits dst 1 opcode
                              opcode-ext x86-64))
       (x86-64 (!flg eflags x86-64)))
      (!!rip
       (cond (dst-index ; write to register
              (!rgfi-size operand-nbytes dst-index result rex x86-64))
             (dst-addr ; write to memory
              (wm-size operand-nbytes dst-addr result x86-64))
             (t ; impossible case, as proved by guard verification
              (prog2$ (er hard ctx
                          "dst-addr and dst-index are both nil.")
```

```
                      x86-64))))))

(defabbrev jump (ctx immediate operand-nbytes ibytes)
  (b* ((operand-nbits (ash operand-nbytes 3))
       (operand-nbits

; Wart: We know already that operand-nbits >= 8.  But the way we are passing
; guards around, that isn't necessarily the case here, or in its caller.

        (max 1 operand-nbits))
       (displacement (sign-extend immediate operand-nbits 64))
       (new-rip (n64+ (next-rip ibytes) displacement))
       ((when (> new-rip *2^48-16*)) ; bad new rip; let's error now
        (!!ms-fresh :new-rip new-rip)))
      (!rip new-rip x86-64)))

(defstep jcc

; Wart: Many opcodes are not yet implemented.

; @@ Deal if necessary with the following for jcc from Intel Vol. 2 JCC
; instruction p. 3-554.
;    If the operand-size attribute is 16, the upper two bytes of the EIP
;    register are cleared, resulting in a maximum instruction pointer size of
;    16 bits.

  (b* ((eflags (flg x86-64))
       (jump-p (case opcode
                 (#x74 ; JE
                  (int= (x86-eflags :zf eflags) 1))
                 (#x75 ; JNE
                  (not (int= (x86-eflags :zf eflags) 1)))
                 (#x7e ; JLE
                  (or (int= (x86-eflags :zf eflags) 1)
                      (not (int= (x86-eflags :sf eflags)
                                 (x86-eflags :of eflags)))))
                 (#x7f ; JG
                  (and (int= (x86-eflags :zf eflags) 0)
                       (int= (x86-eflags :sf eflags)
                             (x86-eflags :of eflags))))
                 (otherwise
                  (er hard? ctx
                      "Impossible case: ~x0"
                      opcode)))))
      (cond
       (jump-p
        (jump ctx immediate operand-nbytes ibytes))
       (t (!!rip x86-64)))))

(defstep jmp
; Wart: Only #xeb and #xe9 are currently implemented.
```

```
  (jump ctx immediate operand-nbytes ibytes))

(defstep lea
  (b* (((mv ?!flg dst-index dst-addr src-index src-addr)
        (src-dst-fields p4 rex opcode ModR/M ModR/M-p sib displacement ibytes
                        *src-dst-fields-ar* operand-nbytes x86-64))
; Note that flg is nil.
       ((when src-index)
        (!!ms-fresh :lea-src-register src-index))
       (operand-nbits (ash operand-nbytes 3))
       (val (logand src-addr (1- (ash 1 operand-nbits)))))
      (!!rip
       (cond (dst-index ; write to register
              (!rgfi-size operand-nbytes dst-index val rex x86-64))
             (t ; impossible case, as proved by guard verification
              (prog2$ (er hard ctx
                          "Dst-index is nil: unexpected!  Dst-addr is: ~x0."
                          dst-addr)
                      x86-64))))))

(defstep leave

; @@ If the operand size is 16 bits, do we move BP to SP or RBP to RSP?  We
; assume the latter in all cases for now, because Intel Vol. 2B p. 4-423 (PUSH
; documentation) says that "... in 64-bit mode, the size of the stack pointer
; is always 64 bits".  But is "the size of the stack pointer" referencing the
; same thing as the "StackAddressSize" mentioned in the LEAVE documentation?
; The documentation for LEAVE in AMD Volume 3 doesn't seem to settle the
; question.

  (b* ((rbp (rgfi *mr-rbp* x86-64))
       (new-rsp (+ rbp operand-nbytes))
       ((when (> new-rsp *2^48*))
        (!!ms-fresh :rbp rbp))
       (val (rm-size operand-nbytes rbp x86-64))
       (x86-64 (!rgfi *mr-rsp* new-rsp x86-64))
       (x86-64 (!rgfi-size operand-nbytes *mr-rbp* val rex x86-64)))
      (!!rip x86-64)))

(defstep mov
  (b* (((mv flg dst-index dst-addr src-index src-addr)
        (src-dst-fields p4 rex opcode ModR/M ModR/M-p sib displacement ibytes
                        *src-dst-fields-ar* operand-nbytes x86-64))
       ((when flg)
        (!!ms-fresh :permission nil))
       (val ; value to be written
        (cond (src-index ; read from register
               (rgfi-size operand-nbytes src-index rex x86-64))
              (src-addr ; read from memory
               (rm-size operand-nbytes src-addr x86-64))
; Otherwise the source operand is an immediate operand.  First we consider the
; special case in which sign extension is necessary.  See the MOV instruction
; table in the Intel manual.

             ((and (int= operand-nbytes 8)
                   (int= opcode #xc7)) ; sign-extend to 64 bits
              (sign-extend immediate 32 64))

; If the special case just above does not apply, then no sign extension is
; necessary, even in the special case of a 64-bit immediate, commented out
; below (because it folds into the final case anyhow).  See AMD Vol. 1
; Sec. 3.2.3.3 ("Immediate Operand Size").

;            ((and (int= operand-nbytes 8)
;                  (int= (logand opcode #xf8)
;                        #xb8)) ;; opcode is B8 through BF
;             immediate) ; 64 bits, so don't sign-extend

             (t immediate))))
     (!!rip
      (cond (dst-index ; write to register
             (!rgfi-size operand-nbytes dst-index val rex x86-64))
            (dst-addr ; write to memory
             (wm-size operand-nbytes dst-addr val x86-64))
            (t ; impossible case, as proved by guard verification
             (prog2$ (er hard ctx
                         "dst-addr and dst-index are both nil.")
                     x86-64))))))

(defstep nop
  (!!rip x86-64))

(defstep pop
  (b* (((when (and (int= opcode #x8f)
                   (not (eql (mrm-reg ModR/M) 0))))
; @@ For opcode #x8f, only 8F/0 is a pop (other cases are not yet
; implemented).

        (!!ms-fresh :unimplemented t :opcode opcode :mrm-reg (mrm-reg ModR/M)))
       (rsp (rgfi *mr-rsp* x86-64))
       (new-rsp (+ rsp operand-nbytes))
       ((when (> new-rsp *2^48*))
        (!!ms-fresh :rsp rsp :new-rsp new-rsp))
       (val (rm-size operand-nbytes rsp x86-64))
       (x86-64

; Quoting Intel Vol. 2B p. 4-337 (POP documentation):

;   If the ESP register is used as a base register for addressing a destination
;   operand in memory, the POP instruction computes the effective address of
;   the operand after it increments the ESP register. For the case of a 16-bit
;   stack where ESP wraps to 0H as a result of the POP instruction, the
;   resulting location of the memory write is processor-family-specific.
;
;   The POP ESP instruction increments the stack pointer (ESP) before data at
;   the old top of stack is written into the destination.

; So, we now update the state before doing the effective address computation
; (see first paragraph above) or writing to the rsp (see second paragraph
; above).   WARNING: undo upon error or exception!

        (!rgfi *mr-rsp* new-rsp x86-64))
       ((mv erp dst-index dst-addr src-index src-addr)
        (src-dst-fields p4 rex opcode ModR/M ModR/M-p sib displacement ibytes
                        *src-dst-fields-ar* operand-nbytes x86-64))
       ((when erp)
        (let ((x86-64 (!rgfi *mr-rsp* rsp x86-64))) ; see WARNING above
          (!!ms-fresh :permission-p nil
                      :rsp rsp
                      :new-rsp new-rsp
                      :operand-nbytes operand-nbytes)))
       ((when (or src-index
                  src-addr
                  (and dst-index dst-addr)
                  (and (int= opcode #x8f)
                       (null dst-index)
                       (null dst-addr))))
; This case is presumably impossible.  @@ Consider proving that.
        (let ((x86-64 (!rgfi *mr-rsp* rsp x86-64))) ; see WARNING above
          (!!ms-fresh :p4 p4 :rex rex :opcode opcode
                      :ModR/M ModR/M :ModR/Mp ModR/M
                      :sib sib :displacement displacement
                      :ibytes ibytes
                      :dst-index dst-index :dst-addr dst-addr
                      :src-index src-index :src-addr src-addr
                      :rsp rsp :new-rsp new-rsp))))
      (!!rip
       (case opcode
         (#x8f
          (cond (dst-index ; write to register
                 (!rgfi-size operand-nbytes dst-index val rex x86-64))
                (dst-addr ; write to memory
                 (wm-size operand-nbytes dst-addr val x86-64))
                (t (prog2$ (er hard ctx
                               "Impossible case!")
                           x86-64))))
         (otherwise ; (int= (logand opcode #x58) #x58)
          (!rgfi-size operand-nbytes dst-index val rex x86-64))))))

(defstep push

; Note, quoting Intel Vol. 2B p. 4-423 (PUSH documentation): "... in 64-bit
; mode, the size of the stack pointer is always 64 bits".

; Quoting the same page as above:

;   The PUSH ESP instruction pushes the value of the ESP register as it existed
;   before the instruction was executed. If a PUSH instruction uses a memory
;   operand in which the ESP register is used for computing the operand
;   address, the address of the operand is computed before the ESP register is
;   decremented.

  (b* (((when (and (int= opcode #xff)
                   (not (eql (mrm-reg ModR/M) 6))))

; @@ For opcode #xff, only FF/6 is a push (other cases are not yet
; implemented).

        (!!ms-fresh :unimplemented t :opcode opcode :mrm-reg (mrm-reg ModR/M)))
       (rsp (rgfi *mr-rsp* x86-64))
       ((when (> rsp *2^48*)) ; written value of rsp = *2^48* seems OK for push

; We need new-rsp + operand-nbytes <= 2^48
; i.e.
; (- rsp operand-nbytes) + operand-nbytes <= 2^48
; i.e.
; (<= rsp *2^48*).

        (!!ms-fresh :rsp rsp))
       (new-rsp (- rsp operand-nbytes))
       ((when (< new-rsp 0))
        (!!ms-fresh :rsp rsp :operand-nbytes operand-nbytes :new-rsp new-rsp))
       ((mv erp dst-index dst-addr src-index src-addr)
        (src-dst-fields p4 rex opcode ModR/M ModR/M-p sib displacement ibytes
                        *src-dst-fields-ar* operand-nbytes x86-64))
       ((when erp)
        (!!ms-fresh :permission-p nil
                    :rsp rsp
                    :new-rsp new-rsp
                    :operand-nbytes operand-nbytes))
       ((when (or dst-index
                  dst-addr
                  (and src-index src-addr)
                  (and (int= opcode #xff)
                       (null src-index)
                       (null src-addr))))
; This case is presumably impossible.  @@ Consider proving that.
        (!!ms-fresh :p4 p4 :rex rex :opcode opcode
                    :ModR/M ModR/M :ModR/Mp ModR/M
                    :sib sib :displacement displacement
                    :ibytes ibytes
                    :dst-index dst-index :dst-addr dst-addr
                    :src-index src-index :src-addr src-addr
                   :rsp rsp :new-rsp new-rsp))
       (val
        (case opcode
          (#xff
           (cond (src-index ; read from register
                  (rgfi-size operand-nbytes src-index rex x86-64))
                 (src-addr ; read from memory
                  (rm-size operand-nbytes src-addr x86-64))
                 (t (prog2$ (er hard ctx
                                "Impossible case!")
                            0))))
          (#x6a

; Quoting Intel Vol. 2B p. 4-423 (PUSH documentation):
;   If the source operand is an immediate and its size is less than the operand
;   size, a sign-extended value is pushed on the stack.

; Note that there is a 3 *decode-operand-nbytes-lst* for this opcode, even
; though we have a one-byte immediate.  Why don't we just push a single byte?
; Quoting page 4-424 Vol. 2B of the Intel manual:

;      IF in 64-bit mode
;        THEN
;          IF operand size = 64
;            THEN
;              RSP <- RSP - 8;
;              Memory[RSP] <- TEMP;
;            ELSE
;              RSP <- RSP - 2;
;              Memory[RSP] <- TEMP;
;          FI;

           (sign-extend immediate 8 (ash operand-nbytes 3)))
          (#x68 immediate)
          (otherwise ; (int= (logand opcode #xf8) #x50)
           (rgfi-size operand-nbytes src-index rex x86-64))))
       (x86-64 (wm-size operand-nbytes new-rsp val x86-64))
       (x86-64 (!rgfi *mr-rsp* new-rsp x86-64)))
      (!!rip x86-64)))

(defstep ret
  (b* ((rsp (rgfi *mr-rsp* x86-64))
       (new-rsp (+ rsp 8))
       ((when (> new-rsp *2^48*))
        (!!ms-fresh :rsp rsp))
       (tos8 (rm64 rsp x86-64))
       ((when (> tos8 *2^48-16*)) ; fail now instead of at next fetch
        (!!ms-fresh :rsp rsp :tos8 tos8))
       (x86-64 (!rgfi *mr-rsp* new-rsp x86-64)))
      (!rip tos8 x86-64)))
```

```
; Step function

(defun x86-64-step-fetch-decode (x86-64)

; We separate out, into this function, the code from our step function that
; sets up our opcode-based case expression.  The reason is that each of those
; opcode step functions has the same guard, and it seems (at least at one point
; in our development) that guard proof obligations were replicated for each
; opcode's step function.  This way, we take care of those proof obligations
; just once.

  (declare (xargs :guard (x86-64p x86-64)
                  :stobjs (x86-64)))
  (b* ((ctx 'x86-64-step)
       ((mv erp itail) ; itail is 15 bytes from the instruction stream
        (x86-64-fetch x86-64))
       ((when erp)
        (mv (!ms-erp :rip (rip x86-64))
            nil nil nil nil nil nil nil nil nil nil nil nil nil))
       (64-bit-modep (64-bit-modep x86-64))
       (cs-dp (cs-dp x86-64))
       ((mv erp p1 p2 p3 p4 rex opcode ModR/M sib displacement immediate
            operand-nbytes ModR/M-p ibytes)
        (x86-64-decode itail 64-bit-modep cs-dp *decode-immediate-ar*
                       *decode-operand-nbytes-ar*))
       ((when erp)
        (mv (!ms-erp :decode-error t :rip (rip x86-64))
            nil nil nil nil nil nil nil nil nil nil nil nil nil))
       ((when p2)

; @@ The presence of p2 is not necessarily erroneous, but we treat it so for
; now.  It can be erroneous -- quoting Intel Vol. 2 Sec. 2.1.1 p. 2-2:

;   Branch hint prefixes (2EH, 3EH) allow a program to give a hint to the
;   processor about the most likely code path for a branch. Use these prefixes
;   only with conditional branch instructions (Jcc). Other use of branch hint
;   prefixes and/or other undefined opcodes with Intel 64 or IA-32 instructions
;   is reserved; such use may cause unpredictable behavior.

        (mv (!ms-erp :prefix-p2 p2 :rip (rip x86-64))
            nil nil nil nil nil nil nil nil nil nil nil nil nil)))
      (mv nil p1 p2 p3 p4 rex opcode ModR/M sib displacement immediate
          operand-nbytes ModR/M-p ibytes)))

(defthm decoded-instruction-p-x86-64-step-fetch-decode
  (implies (x86-64p x86-64)
           (mv-let (erp p1 p2 p3 p4 rex opcode ModR/M sib displacement
                        immediate operand-nbytes ModR/M-p ibytes)
                   (x86-64-step-fetch-decode x86-64)
                   (declare (ignore ibytes))
                   (implies (not erp)
                           (decoded-instruction-p
                            p1 p2 p3 p4 rex opcode modr/m sib
                            displacement immediate operand-nbytes
                            modr/m-p))))
  :hints (("Goal"
           :restrict
           ((decoded-instruction-p-x86-64-decode
             ((64-bit-modep (64-bit-modep x86-64))
              (cs-dp (cs-dp x86-64))
              (decode-immediate-ar *decode-immediate-ar*)
              (decode-operand-nbytes-ar
               *decode-operand-nbytes-ar*)
              (instr (nth 1 (x86-64-fetch x86-64))))))))
  :rule-classes nil)

(encapsulate
 ()
 (local (in-theory (e/d (x86-64-decode get-instruction-byte)
                        (x86-64-decode-displacement
                         x86-64-decode-immediate
                         x86-64-decode-operand-nbytes
                         x86-64-decode-prefix
                         x86-64-decode-rex x86-64-decode-sib))))
 (defthm natp-ibytes-x86-64-step-fetch-decode
   (implies (x86-64p x86-64)
            (mv-let (erp p1 p2 p3 p4 rex opcode ModR/M sib displacement
                         immediate operand-nbytes ModR/M-p ibytes)
                    (x86-64-step-fetch-decode x86-64)
                    (declare (ignore p1 p2 p3 p4 rex opcode ModR/M sib
                                     displacement immediate operand-nbytes
                                     ModR/M-p))
                    (implies (not erp)
                             (natp ibytes))))
   :rule-classes nil))

(defmacro x86-64-step-cases ()
  (x86-64-step-cases-fn
   *op-alist*
   '((otherwise (x86-64-step-unimplemented opcode x86-64)))))

(defun x86-64-step (x86-64)

; Returns an x86-64 obtained by executing one instruction.

; @@ Need to deal with address translation (virtual to physical).

  (declare (xargs :guard (x86-64p x86-64)
                  :guard-hints
                  (("Goal"
                    :in-theory
                    (disable x86-64-step-fetch-decode)
```

```
                   :use (decoded-instruction-p-x86-64-step-fetch-decode
                         natp-ibytes-x86-64-step-fetch-decode)))
                 :stobjs (x86-64)))
  (mv-let
   (erp p1 p2 p3 p4 rex opcode ModR/M sib displacement immediate operand-nbytes
        ModR/M-p ibytes)
   (x86-64-step-fetch-decode x86-64)
   (cond (erp (!ms erp x86-64))
         (t (x86-64-step-cases)))))

(defthm decoded-instruction-p-x86-64-step-fetch-decode-rewrite
  (implies (and (x86-64p x86-64)
                (equal erp (nth 0 (x86-64-step-fetch-decode x86-64)))
                (equal p1 (nth 1 (x86-64-step-fetch-decode x86-64)))
                (equal p2 (nth 2 (x86-64-step-fetch-decode x86-64)))
                (equal p3 (nth 3 (x86-64-step-fetch-decode x86-64)))
                (equal p4 (nth 4 (x86-64-step-fetch-decode x86-64)))
                (equal rex (nth 5 (x86-64-step-fetch-decode x86-64)))
                (equal opcode (nth 6 (x86-64-step-fetch-decode x86-64)))
                (equal ModR/M (nth 7 (x86-64-step-fetch-decode x86-64)))
                (equal sib (nth 8 (x86-64-step-fetch-decode x86-64)))
                (equal displacement (nth 9 (x86-64-step-fetch-decode x86-64)))
                (equal immediate (nth 10 (x86-64-step-fetch-decode x86-64)))
                (equal operand-nbytes (nth 11 (x86-64-step-fetch-decode x86-64)))
                (equal ModR/M-p (nth 12 (x86-64-step-fetch-decode x86-64))))
           (implies (not erp)
                    (decoded-instruction-p
                     p1 p2 p3 p4 rex opcode modr/m sib
                     displacement immediate operand-nbytes
                     modr/m-p)))
  :hints (("Goal"
           :restrict
           ((decoded-instruction-p-x86-64-decode
             ((64-bit-modep (64-bit-modep x86-64))
              (cs-dp (cs-dp x86-64))
              (decode-immediate-ar *decode-immediate-ar*)
              (decode-operand-nbytes-ar
               *decode-operand-nbytes-ar*)
              (instr (nth 1 (x86-64-fetch x86-64)))))))))

(defthm x86-64p-x86-64-step
  (implies (and (x86-64p x86-64)
                (not (nth *ms* x86-64)))
           (x86-64p (x86-64-step x86-64))))

(in-theory (disable x86-64-step))

(defun x86-64-run (n x86-64)
; Returns an x86-64 obtained by executing n instructions (or until halting).
  (declare (xargs :guard (and (natp n)
                              (x86-64p x86-64))
```

```
                  :stobjs (x86-64)))
  (cond ((ms x86-64)
         x86-64)
        ((zp n)
         (let ((ctx 'x86-64-run))
           (!!ms-fresh :timeout t)))
        (t (let ((x86-64 (x86-64-step x86-64)))
             (x86-64-run (1- n) x86-64)))))

(defthm x86-64p-x86-64-run
  (implies (x86-64p x86-64)
           (x86-64p (x86-64-run n x86-64))))

(defun x86-64-run-steps1 (n n0 x86-64)
; Returns an x86-64 obtained by executing n instructions (or until halting).
  (declare (xargs :guard (and (natp n)
                              (acl2-numberp n0)
                              (x86-64p x86-64))
                  :stobjs (x86-64)))
  (cond ((ms x86-64)
         (mv (- n0 n) x86-64))
        ((zp n)
         (let* ((ctx 'x86-64-run)
                (x86-64 (!!ms-fresh :timeout t)))
           (mv (- n0 n) x86-64)))
        (t (let ((x86-64 (x86-64-step x86-64)))
             (x86-64-run-steps1 (1- n) n0 x86-64)))))

(defun x86-64-run-steps (n x86-64)
; Returns an x86-64 obtained by executing n instructions (or until halting).
  (declare (xargs :guard (and (natp n)
                              (x86-64p x86-64))
                  :stobjs (x86-64)))
  (x86-64-run-steps1 n n x86-64))

(defthm x86-64-run-steps1-is-x86-64-run
  (equal (nth 1 (x86-64-run-steps1 n n0 x86-64))
         (x86-64-run n x86-64)))

(defthm x86-64-run-steps-is-x86-64-run
  (equal (nth 1 (x86-64-run-steps n x86-64))
         (x86-64-run n x86-64)))
```

17 ========== File tools/script.lisp ==========

```
; NOTE: This script

; In order to test the fibonacci program, execute the following in the ACL2
; loop:

; (ld "script.lisp")
; This will use the x86-64 model to run fib on input 8, but you can then run
; with other inputs, e.g.:

; (run-fib 20 x86-64)

; If you want to get out of raw mode after LDing this file:

; (set-raw-mode nil)

; If you want to trace the interpreter, do the following.

; (set-raw-mode nil)
; (set-print-base 16 state)
; (include-book "xtrace")
; (xtrace)
; (set-raw-mode-on!)

(defun fib (n)
  (declare (xargs :guard (natp n)))
  (cond ((zp n) 0)
        ((eql n 1) 1)
        (t (+ (fib (- n 1)) (fib (- n 2))))))

(include-book "../x86-64/x86")
(include-book "populate-stobj-with-prog-bytes" :uncertified-okp t)
(ld "fib-addr-byte.lisp")
(assign xrun-limit 100000000)

(set-raw-mode-on!)

(defun !rgfi-from-alist (alist x86-64)
  (cond ((endp alist) x86-64)
        (t (let ((x86-64 (!rgfi (caar alist) (cdar alist) x86-64)))
             (!rgfi-from-alist (cdr alist) x86-64)))))

(defun setup-for-run (binary start-address reg-alist halt-address x86-64)

; Set up for a run.

  (!ms nil x86-64) ; in case this isn't the first run
  (load-bytes-into-memory binary x86-64)
  (wm08 halt-address #xf4 x86-64)
  (!rip start-address x86-64)
  (!rgfi-from-alist reg-alist x86-64)
  (!rgfi *mr-rsp* *2^48* x86-64))

(defun run-fib (input x86-64 &aux (ctx 'run-fib))
  (setup-for-run *fib-binary*
                 #x100000e67 ; start-address
                 (list (cons *mr-rax* input))
```

```
                #x100000e70 ; halt-address
                x86-64)
  (mv-let
   (fib-steps x86-64)
   (time$ (x86-64-run-steps (@ xrun-limit) x86-64))
   (state-free-global-let*
    ((print-base 10))
    (cond ((not (equal (ms x86-64)
                       '((X86-64-STEP-HLT :RIP #x100000E70
                                          :HLT :LEGAL-HALT))))
           (er soft ctx
               "~|(ms x86-64) = ~x0"
               (ms x86-64)))
          (t (let ((expected (fib input)))
               (cond
                ((equal (rgfi *mr-rax* x86-64)
                        expected)
                 (pprogn
                  (fmx "(fib ~x0) was correctly computed as ~x1 (~x2 steps)~|"
                       input
                       expected
                       fib-steps)
                  (value t)))
                (t (er soft ctx
                       "(fib ~x0) = ~x1, but rax is ~x2"
                       input
                       expected
                       (rgfi *mr-rax* x86-64)))))))))
  nil)

; Feel free to run with other inputs besides 8!
(run-fib 8 x86-64)

; A second run if we choose....
(er-let* ((str (getenv$ "RUN_FIB_N" state)))
  (cond (str (run-fib (decimal-string-to-number str (length str) 0)
                      x86-64)
             (value nil))
        (t (value nil))))
```

18 ==== File tools/populate-stobj-with-prog-bytes.lisp ====

```
; Shilpi Goel

(in-package "ACL2")

; Load bytes into memory:

; The lisp file created using the shell script "addr-byte" defines the list of
; address-byte pairs of the program binary as a constant. This list is then
; used as an input to the function load-bytes-into-memory, which populates the
; memory of the x86-64 stobj accordingly.

; Program Binary Recognizer --- a list of (<address> . <byte>) pairs.

(defun addr-bytes-alistp (alst)
  (declare (xargs :guard t))
  (if (atom alst)
      (equal alst nil)
    (if (atom (car alst))
        nil
      (let ((addr (caar alst))
            (byte    (cdar alst))
            (rest   (cdr  alst)))
        (and (n48p addr)
             (n08p byte)
             (addr-bytes-alistp rest))))))

(defun load-bytes-into-memory (addr-bytes-lst x86-64)
  (declare (xargs :guard (and (x86-64p x86-64)
                              (true-listp addr-bytes-lst)
                              (addr-bytes-alistp addr-bytes-lst))
                  :stobjs (x86-64)))
  (cond ((endp addr-bytes-lst) x86-64)
        (t
         (let* ((addr (caar addr-bytes-lst))
                (byte (cdar addr-bytes-lst))
                (x86-64 (wm08 addr
                              byte
                              x86-64)))
           (load-bytes-into-memory (cdr addr-bytes-lst) x86-64)))))
```

19 ========== File tools/fib-addr-byte.lisp ==========

```
(defconst *fib-binary*
  (list
   (cons #x100000dc0 #x6a)
   (cons #x100000dc1 #x00)
   (cons #x100000dc2 #x48)
   (cons #x100000dc3 #x89)
   (cons #x100000dc4 #xe5)
   (cons #x100000dc5 #x48)
   (cons #x100000dc6 #x83)
   (cons #x100000dc7 #xe4)
   (cons #x100000dc8 #xf0)
   (cons #x100000dc9 #x48)
   (cons #x100000dca #x8b)
   (cons #x100000dcb #x7d)
   (cons #x100000dcc #x08)
   (cons #x100000dcd #x48)
   (cons #x100000dce #x8d)
   (cons #x100000dcf #x75)
```

```
(cons #x100000dd0 #x10)
(cons #x100000dd1 #x89)
(cons #x100000dd2 #xfa)
(cons #x100000dd3 #x83)
(cons #x100000dd4 #xc2)
(cons #x100000dd5 #x01)
(cons #x100000dd6 #xc1)
(cons #x100000dd7 #xe2)
(cons #x100000dd8 #x03)
(cons #x100000dd9 #x48)
(cons #x100000dda #x01)
(cons #x100000ddb #xf2)
(cons #x100000ddc #x48)
(cons #x100000ddd #x89)
(cons #x100000dde #xd1)
(cons #x100000ddf #xeb)
(cons #x100000de0 #x04)
(cons #x100000de1 #x48)
(cons #x100000de2 #x83)
(cons #x100000de3 #xc1)
(cons #x100000de4 #x08)
(cons #x100000de5 #x48)
(cons #x100000de6 #x83)
(cons #x100000de7 #x39)
(cons #x100000de8 #x00)
(cons #x100000de9 #x75)
(cons #x100000dea #xf6)
(cons #x100000deb #x48)
(cons #x100000dec #x83)
(cons #x100000ded #xc1)
(cons #x100000dee #x08)
(cons #x100000def #xe8)
(cons #x100000df0 #x5c)
(cons #x100000df1 #x00)
(cons #x100000df2 #x00)
(cons #x100000df3 #x00)
(cons #x100000df4 #x89)
(cons #x100000df5 #xc7)
(cons #x100000df6 #xe8)
(cons #x100000df7 #xa9)
(cons #x100000df8 #x00)
(cons #x100000df9 #x00)
(cons #x100000dfa #x00)
(cons #x100000dfb #xf4)
(cons #x100000dfc #x90)
(cons #x100000dfd #x90)
(cons #x100000dfe #x90)
(cons #x100000dff #x90)
(cons #x100000e00 #x55)
(cons #x100000e01 #x48)
(cons #x100000e02 #x89)
```

```
(cons #x100000e03 #xe5)
(cons #x100000e04 #x41)
(cons #x100000e05 #x54)
(cons #x100000e06 #x53)
(cons #x100000e07 #x89)
(cons #x100000e08 #xfb)
(cons #x100000e09 #x45)
(cons #x100000e0a #x31)
(cons #x100000e0b #xe4)
(cons #x100000e0c #x85)
(cons #x100000e0d #xff)
(cons #x100000e0e #x7e)
(cons #x100000e0f #x27)
(cons #x100000e10 #x41)
(cons #x100000e11 #xb4)
(cons #x100000e12 #x01)
(cons #x100000e13 #x83)
(cons #x100000e14 #xff)
(cons #x100000e15 #x01)
(cons #x100000e16 #x74)
(cons #x100000e17 #x1f)
(cons #x100000e18 #x45)
(cons #x100000e19 #x31)
(cons #x100000e1a #xe4)
(cons #x100000e1b #xeb)
(cons #x100000e1c #x08)
(cons #x100000e1d #x0f)
(cons #x100000e1e #x1f)
(cons #x100000e1f #x00)
(cons #x100000e20 #x83)
(cons #x100000e21 #xfb)
(cons #x100000e22 #x01)
(cons #x100000e23 #x74)
(cons #x100000e24 #x1a)
(cons #x100000e25 #x8d)
(cons #x100000e26 #x7b)
(cons #x100000e27 #xff)
(cons #x100000e28 #xe8)
(cons #x100000e29 #xd3)
(cons #x100000e2a #xff)
(cons #x100000e2b #xff)
(cons #x100000e2c #xff)
(cons #x100000e2d #x83)
(cons #x100000e2e #xeb)
(cons #x100000e2f #x02)
(cons #x100000e30 #x41)
(cons #x100000e31 #x01)
(cons #x100000e32 #xc4)
(cons #x100000e33 #x85)
(cons #x100000e34 #xdb)
(cons #x100000e35 #x7f)
```

```
(cons #x100000e36 #xe9)
(cons #x100000e37 #x44)
(cons #x100000e38 #x89)
(cons #x100000e39 #xe0)
(cons #x100000e3a #x5b)
(cons #x100000e3b #x41)
(cons #x100000e3c #x5c)
(cons #x100000e3d #xc9)
(cons #x100000e3e #xc3)
(cons #x100000e3f #x41)
(cons #x100000e40 #xff)
(cons #x100000e41 #xc4)
(cons #x100000e42 #x44)
(cons #x100000e43 #x89)
(cons #x100000e44 #xe0)
(cons #x100000e45 #x5b)
(cons #x100000e46 #x41)
(cons #x100000e47 #x5c)
(cons #x100000e48 #xc9)
(cons #x100000e49 #xc3)
(cons #x100000e4a #x66)
(cons #x100000e4b #x0f)
(cons #x100000e4c #x1f)
(cons #x100000e4d #x44)
(cons #x100000e4e #x00)
(cons #x100000e4f #x00)
(cons #x100000e50 #x55)
(cons #x100000e51 #x48)
(cons #x100000e52 #x89)
(cons #x100000e53 #xe5)
(cons #x100000e54 #x53)
(cons #x100000e55 #x48)
(cons #x100000e56 #x83)
(cons #x100000e57 #xec)
(cons #x100000e58 #x08)
(cons #x100000e59 #x83)
(cons #x100000e5a #xff)
(cons #x100000e5b #x02)
(cons #x100000e5c #x75)
(cons #x100000e5d #x2a)
(cons #x100000e5e #x48)
(cons #x100000e5f #x8b)
(cons #x100000e60 #x7e)
(cons #x100000e61 #x08)
(cons #x100000e62 #xe8)
(cons #x100000e63 #x37)
(cons #x100000e64 #x00)
(cons #x100000e65 #x00)
(cons #x100000e66 #x00)
(cons #x100000e67 #x89)
(cons #x100000e68 #xc3)
```

```
(cons #x100000e69 #x89)
(cons #x100000e6a #xc7)
(cons #x100000e6b #xe8)
(cons #x100000e6c #x90)
(cons #x100000e6d #xff)
(cons #x100000e6e #xff)
(cons #x100000e6f #xff)
(cons #x100000e70 #x89)
(cons #x100000e71 #xc2)
(cons #x100000e72 #x89)
(cons #x100000e73 #xde)
(cons #x100000e74 #x48)
(cons #x100000e75 #x8d)
(cons #x100000e76 #x3d)
(cons #x100000e77 #x50)
(cons #x100000e78 #x00)
(cons #x100000e79 #x00)
(cons #x100000e7a #x00)
(cons #x100000e7b #x31)
(cons #x100000e7c #xc0)
(cons #x100000e7d #x48)
(cons #x100000e7e #x83)
(cons #x100000e7f #xc4)
(cons #x100000e80 #x08)
(cons #x100000e81 #x5b)
(cons #x100000e82 #xc9)
(cons #x100000e83 #xe9)
(cons #x100000e84 #x22)
(cons #x100000e85 #x00)
(cons #x100000e86 #x00)
(cons #x100000e87 #x00)
(cons #x100000e88 #x48)
(cons #x100000e89 #x8d)
(cons #x100000e8a #x3d)
(cons #x100000e8b #x27)
(cons #x100000e8c #x00)
(cons #x100000e8d #x00)
(cons #x100000e8e #x00)
(cons #x100000e8f #xe8)
(cons #x100000e90 #x1c)
(cons #x100000e91 #x00)
(cons #x100000e92 #x00)
(cons #x100000e93 #x00)
(cons #x100000e94 #xbf)
(cons #x100000e95 #x01)
(cons #x100000e96 #x00)
(cons #x100000e97 #x00)
(cons #x100000e98 #x00)
(cons #x100000e99 #xe8)
(cons #x100000e9a #x06)
(cons #x100000e9b #x00)
```

```
(cons #x100000e9c #x00)
(cons #x100000e9d #x00)
(cons #x100001000 #x00)
(cons #x100001001 #x00)
(cons #x100001002 #x00)
(cons #x100001003 #x00)
(cons #x100001004 #x01)
(cons #x100001005 #x00)
(cons #x100001006 #x00)
(cons #x100001007 #x00)
(cons #x100001008 #x58)
(cons #x100001009 #x10)
(cons #x10000100a #x00)
(cons #x10000100b #x00)
(cons #x10000100c #x01)
(cons #x10000100d #x00)
(cons #x10000100e #x00)
(cons #x10000100f #x00)
(cons #x100001010 #x60)
(cons #x100001011 #x10)
(cons #x100001012 #x00)
(cons #x100001013 #x00)
(cons #x100001014 #x01)
(cons #x100001015 #x00)
(cons #x100001016 #x00)
(cons #x100001017 #x00)
(cons #x100001018 #x68)
(cons #x100001019 #x10)
(cons #x10000101a #x00)
(cons #x10000101b #x00)
(cons #x10000101c #x01)
(cons #x10000101d #x00)
(cons #x10000101e #x00)
(cons #x10000101f #x00)
(cons #x100001020 #x70)
(cons #x100001021 #x10)
(cons #x100001022 #x00)
(cons #x100001023 #x00)
(cons #x100001024 #x01)
(cons #x100001025 #x00)))
```